**University of Padua**

Mathematics Department "Tullio Levi-Civita"

Master Degree in Computer Science

# Greedy Approach to Compute Alignments of Process Models and Event Logs

*Master Thesis*

*Supervisor*

Prof.Massimiliano de Leoni

*Student*

Sofia Chiarello

Academic Year 2022-2023

*"Si manana soy yo, mamà, si manana no vuelvo, destruyelo todo"*
— Cristina Torres Càceres

To my parents

# Summary

In Process Mining, computing alignments is a Conformance Checking technique to compare a process model with an event log of the same process to pinpoint difference between how the model would prescribe the process to be executed, and how the event log states the process has been executed. The complexity of this problem is naturally exponential with respect to the size of the model, and benefits can be achieved using divide-and-conquer approaches: the model is decomposed into small fragment for which we can compute alignments. This thesis compares the time to compute alignments using the traditional approaches and our decomposition-based approaches to identify the possible benefits. The results are also compared with different approaches based on process-model decompositions.

# Acknowledgments

*Innanzitutto, vorrei esprimere la mia gratitudine al Prof. Massimiliano de Leoni, relatore della mia tesi, per l'aiuto e il sostegno durante lo sviluppo di questo progetto. Inoltre, grazie alle sue lezioni, mi ha permesso di appassionarmi a questa materia.*

*Desidero ringraziare con affetto i miei genitori, Patrizia e Primo, che mi hanno supportato sempre durante il mio percorso di studi e nella vita, questo traguardo lo devo a loro. Mio fratello Matteo, il quale è stato una spalla su cui contare durante tutta la mia crescita.*

*Ho inoltre desiderio di ringraziare tutti i miei amici, con i quali ho condiviso molte esperienze. In particolare Giada, la quale è passata da semplice compagna di progetto a una delle mie migliori amiche.*

*Alla mia grande famiglia va un grande grazie, soprattutto alle mie due nonne, Luisa e Pia, le quali hanno contribuito a crescermi e a rendermi la donna che sono oggi.*

*Ultimo ma non per importanza, va un caloroso ringraziamento al mio compagno Fabio, il quale è stato al mio fianco in ogni momento, la sua presenza è stata fondamentale per la mia motivazione e per la riuscita di questo traguardo.*

*Padova, December 2023* Sofia Chiarello

# Contents

**1 Introduction** **1**

**2 Process Mining Fundamentals** **5**

2.1 Process Mining . . . . . . . . . . . . 5

2.2 Notations and Definitions . . . . . . . . . . . . 6

2.2.1 Basic Notations . . . . . . . . . . . . 6

2.2.2 Petri Net . . . . . . . . . . . . 6

2.2.3 Event Logs . . . . . . . . . . . . 9

2.3 Tools for Process Mining . . . . . . . . . . . . 10

2.3.1 ProM . . . . . . . . . . . . 10

2.3.2 XES . . . . . . . . . . . . 10

**3 Conformance Checking** **13**

3.1 Preliminaries . . . . . . . . . . . . 13

3.1.1 Token-Replay Conformance Checking . . . . . . . . . . . . 14

3.1.2 Alignment-Based Conformance Checking . . . . . . . . . . . . 15

3.1.3 Conformance Checking Algorithm . . . . . . . . . . . . 19

3.2 Decomposition-Based Approach . . . . . . . . . . . . 21

3.2.1 Decomposition in Conformance Checking . . . . . . . . . . . . 22

3.2.2 Decomposition Algorithms for Splitting Petri Nets . . . . . . . 27

3.2.3 Decomposition-Based Algorithms for Conformance Checking . 31

**4 Greedy Approach to Compute Alignments of Process Models and Events Logs** **37**

4.1 Algorithm . . . . . . . . . . . . 37

4.1.1 Improved Algorithm . . . . . . . . . . . . 42

4.2 Implementation . . . . . . . . . . . . 44

4.2.1 Splitting Petri Net and Event Log . . . . . . . . . . . . 44

4.2.2 Computing Subalignments . . . . . . . . . . . . 45

4.2.3 Checking Conflicts . . . . . . . . . . . . 49

4.2.4 Merging Subalignments . . . . . . . . . . . . 51

4.3 Implementation Problems Encountered . . . . . . . . . . . . 51

4.3.1 Empty Subtraces . . . . . . . . . . . . 51

4.3.2 Memory Issue . . . . . . . . . . . . 53

**5 Experiments** **55**

5.1 Preliminary Steps . . . . . . . . . . . . 55

5.2 Results . . . . . . . . . . . . 57

5.2.1 Noiseless Scenario . . . . . . . . . . . . 57

5.2.2 Noisy Scenario . . . . . . . . . . 58
5.3 Discussion and Future Works . . . . . . . . . . 64

**6 Conclusions 67**

**References 69**

# List of Figures

1.1 Example of a huge process model. . . . . . . . . . . . . . . . . . . . . . . 2

2.1 Positioning of the three main types of Process Mining: Discovery, Conformance Checking, Enhancement. . . . . . . . . . . . . . . . . . . . . . . 6
2.2 The elevator behavior modeled as Petri net. Source: [2] . . . . . . . . 7
2.3 An example of an event log. Source: [3] . . . . . . . . . . . . . . . . . . 9
2.4 Meta-model of XES expressed using UML class diagrams. Source: [3] 11

3.1 Replaying of trace $\sigma_1 = \langle a, d, c, e, h \rangle$ on the net $N$. The various stages of replay are represented with the according four counters: p (produced tokens), c (consumed tokens), m (missing tokens), and r (remaining tokens). Source: [3] . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
3.2 Petri Net $N_2$. Source: [3] . . . . . . . . . . . . . . . . . . . . . . . . . 17
3.3 Visualization of the result of *"Replay a log on Petri Net for conformance analysis"* plug-in of ProM software. We can see for each line, the alignment corresponding to a trace. . . . . . . . . . . . . . . . . . . . . . . 20
3.4 *Synchronous product* Petri net. In gray is pictured the original net; in yellow the transitions corresponding to activities in the trace, and in green the synchronous transitions. . . . . . . . . . . . . . . . . . . . . . . 20
3.5 Configuration of the parameters to build alignments in the plug-in *"Replay a log on Petri Net for conformance analysis"*. Note that the standard configuration uses the *Iterative A* technique*. . . . . . . . . . 21
3.6 Petri net with $n$ parallel activities. . . . . . . . . . . . . . . . . . . . . 22
3.7 A possible decomposition of the net in Figure 3.6. . . . . . . . . . . . 22
3.8 System net $SN_1$. Source: [1] . . . . . . . . . . . . . . . . . . . . . . . 23
3.9 Valid decomposition $D_1$ of the system net $SN_1$. Source: [1] . . . . . . 23
3.10 Labeled Petri net. Source: [7] . . . . . . . . . . . . . . . . . . . . . . . 27
3.11 Maximal decomposition of the system net shown in 3.10. Source: [7] . 28
3.12 A Petri net, its workflow graph and the RPST and SESE decomposition. (a) Petri net. (b) Workflow Graph and SESE decomposition. (c) RPST. Source: [7] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
3.13 Example of bridging technique. (a) Petri net with a SESE decomposition. (b) A SESE decomposition. (c) A SESE decomposition with bridging. Source: [9] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
3.14 Overview of the Recomposing Replay Algorithm. Source: [1] . . . . . 33
3.15 New decomposition $D_2$ after one recomposing interation. Source: [1] . 33

4.1 System Net $SN_2$ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

4.2 Decomposition $D_2$ of the Petri net $SN_2$ depicted in Figure 4.1 . . . . 39
4.3 Visualization of the result of our approach implemented in ProM. Here the moves of all alignments are projected into occurrences of activities. 42
4.4 Other visualization of the result of our approach implemented in ProM. The plug-in provides the alignments computed using our approach. The green moves are synchronous moves, the purple ones are model moves instead yellow ones are log moves. . . . . . . . . . . . . . . . . . . . . . 42
4.5 Visualization of ProM to select our algorithm . . . . . . . . . . . . . . 44
4.6 Visualization of the configuration for splitting a Petri net from the plug-in *"Split Accepting Petri Net"*. On the right is possible to select which decomposition strategy use. . . . . . . . . . . . . . . . . . . . . . 45
4.7 Visualization of the starting of the plug-in *"Replay a Log on Petri Net for All Optimal Alignments"* in Prom . . . . . . . . . . . . . . . . . . . 46
4.8 Visualization of the result of *"Replay a Log on Petri Net for All Optimal Alignments"* plug-in. It is computed all the possible alignments of trace *case_ 9* from fitness 1 to fitness equal to 0.8 . . . . . . . . . . . . . . . 46
4.9 Array of subalignments of trace $\gamma_0$ and Petri net $SN_2$. Each column corresponds to a subnet. In the first row are present the optimal subalignments computed. . . . . . . . . . . . . . . . . . . . . . . . . . 48
4.10 Array of subalignments of trace $\gamma_0$ and Petri net $SN_2$ after one iteration, i.e. computing the next-best subalignment operation. The subalignments of subnet number four is selected, so its next-best is stored in the second row. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
4.11 Petri net $SN_3$ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
4.12 Decomposition $D_3$ of System Net $SN_3$ in picture 4.11. . . . . . . . . . 49
4.13 Representation of algorithm to check conflicts. In the first iteration, it checks the *d synchronous move*. As it is present also in the next subalignment, the cursor is updated to this position. In the following iteration, the *d log move* is checked. There is a conflict as the next alignment presents a *d synchronous move*. . . . . . . . . . . . . . . . . 50
4.14 Example of three subnets of the System net $SN_2$ modified in order to solve the issue of empty alignments. In (1) it is pictured the initial subnet; in (2) the final subnet and in (3) the generic case. The added part is highlighted. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52

5.1 Petri Net P0 of dataset. It presents 206 activities and 92 gateways. . . 56
5.2 Visualization of the noise configuration of plug-in *"Add, Swap and Remove Events"* implemented in ProM. . . . . . . . . . . . . . . . . . . 57
5.3 Graph representing computation time for each algorithm in noiseless scenario. The processes are ordered by number of activities. For each of them the time expressed in seconds are compared with three methods. 59
5.4 Graphs representing percentage of pseudo-alignments computed by Greedy Approach and Recomposition Replay. The first graph presents the traces with 6% of noise, the second one presents the traces with 14% of noise. With complex nets and more noise, Recomposition Replay results touch almost the 100% of pseudo-alignments. . . . . . . . . . . 61
5.5 Graph representing computation time for each algorithm in a scenario with noise equal to 6%. The processes are ordered by number of activities. For each of them the time expressed in seconds are compared with three methods. Note that times are shown in a logarithmic scale. . . . . . 61

5.6 Graph representing fitness values for each algorithm in a scenario with noise equal to 6%. The processes are ordered by number of activities. The fitness are computed for decomposition approaches using just the traces which produce alignment (i.e. exact fitness can be calculated). Note that all three techniques return the same results for each model. 62
5.7 Graph representing computation time for each algorithm in a scenario with noise equal to 14%. The processes are ordered by number of activities. For each of them the time expressed in seconds are compared with three methods. Note that times are shown in a logarithmic scale. 63
5.8 Graph representing fitness values for each algorithm in a scenario with noise equal to 14%. The processes are ordered by number of activities. The fitness are computed for decomposition approaches using just the traces which produce alignment (i.e. exact fitness can be calculated). . 64
5.9 Naive representation of searching tree to check all possible subalignments combination. The subalignments are depicted using letters. In the root are present the combination of just optimal subalignments. Then subalignment A, B and C are selected sequentially to compute their next-best alignments. The search goes on by selecting one subalignments and computing its next-best. . . . . . . . . . . . . . . . . . . . . . . . 66

# List of Tables

5.1 Characteristics of the synthetic nets. . . . . . . . . . . . . . . . . . . . . 56
5.2 Results about Greedy Approach for Decomposed Replay in noiseless scenario. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
5.3 Results about computation time and fitness between three algorithms in noiseless scenario. . . . . . . . . . . . . . . . . . . . . . . . . . . 59
5.4 Results about Greedy Approach for Decomposed Replay . . . . . . . . 60
5.5 Results about computation time and fitness between three algorithms with noise=6% . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
5.6 Results about computation time and fitness between three algorithms with noise=14% . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63

# Chapter 1

# Introduction

Nowadays, informatics systems are rapidly spreading and their expanding capabilities are well characterizes by Moore's law. Moore[1] predicted in 1965 that the number of components in integrated circuits would double every year. The growth has indeed be exponential in the last fifty years. These improvements have resulted in an enormous growth of the digital universe. Organizations of all kinds embrace this revolution and adopt information systems not only to simplify work procedures but also to collect *data*.

Data are generated by every operation we perform daily. The data recorded by organization devices may be linked to the *new oil*. Data represents a powerful tool to improve firms' businesses and processes. The growth of the digital universe is not only related to static data, but also makes it possible to record and analyze events about processes. This represents a complete history of past operations performed and can provide insights into what needs improvement and how we can achieve that. However, raw data is not useful, as its information is masked by its complexity. We need a way to exploit this data in a meaningful way, for example, to provide insights, identify bottlenecks, anticipate problems, record policy violations, recommend countermeasures, and streamline processes. Process Mining discipline aims to do exactly this.

Process Mining has emerged as a highly active research area in recent years, and its interest continues to grow. It provides various analysis techniques capable of handling data and offering visual representations and insights for interpretation and understanding. Process Mining is often regarded as an intersection of multiple studies fields, including Computer Science, Data Mining, Economics, and more. It is frequently seen as a link connecting Data Science and Process Science.

One of the significant area within Process Mining is *Conformance Checking*. Conformance Checking is the problem of detecting deviations between how processes are executed in reality and how processes are allowed to be performed based on norms, regulations, and protocols. *Event logs* serve as the starting point of Process Mining in general, and Conformance Checking in particular: they record the events that occurred in the various executions of processes, while the set of allowed executions are summarized in process models. By comparing an event log with the process model of the same process, differences in behaviors can be detected. Intuitively, Conformance Checking serves as an essential tool for identifying problems.

[1]Gordon Moore was an American businessman and co-founded of Intel Corporation

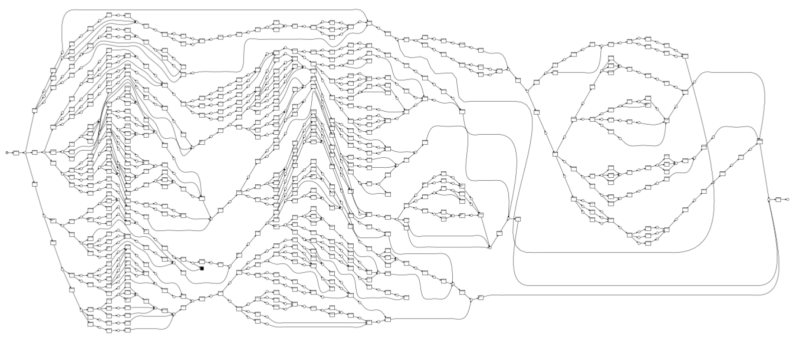


**Figure 1.1:** Example of a huge process model.

Conformance Checking techniques are exponential with the respect to the size of the model, making it challenging to scale when dealing with very large models. Figure 1.1 provides an example of the considerable size that a process model can reach.

In response to this complexity challenge, various new approaches have been developed. One category of such approaches involves decomposing the process model into smaller parts for which Conformance Checking is performed: in this way, one single exponential problem is decomposed into a collection of smaller problems, with evident benefits for the complexity. While early works related to decomposition-based Conformance Checking have shown satisfactory results, there is still room for improvement, especially in cases where number of sub-models is unavoidably large. This thesis introduces a novel technique that utilizes a decomposition-based approach to perform Conformance Checking, aiming to address some weaknesses found in state-of-the-art methods and serve as a valid alternative to them.

Once we implemented our novel algorithm, we conducted a series of experiments to assess its effectiveness compared to existing methods. The experiments were conducted in two scenarios: *noiseless*, where traces perfectly fit the model, and *noisy*, where there are discrepancies between the modeled and observed behaviors. We evaluated the overall fitness value and computational times using a dataset of ten different processes computed by three different algorithms: our novel approach, the existing monolithic procedure and the more recent decomposition-based approach called *Recomposing Replay* algorithm. The results indicate that our approach could serve as a valid alternative to the decomposition-based method, particularly in noisy scenario, as it demonstrated faster computational times. However, it is important to note that the monolithic approach still outperforms both algorithms in terms of speed and accuracy. This superiority is attributed to the utilization of a really recent version of the monolithic algorithm, which has been significantly improved, rendering existing decomposition-based experiments obsolete. Nevertheless, we have actively explored ways to enhance our approach for better results. We are confident that in very complex models, its computational times could be further reduced compared to traditional algorithms.

## Document Outline

The present document is organized as follows:

**Chapter 2:** to encourage a better understanding, this chapter provides Process Mining fundamentals such as process model and event logs.

**Chapter 3:** this chapter explains in detail the Conformance Checking problems, providing examples of the existing techniques. In the second part of the chapter, we go in-depth with decomposition-based approach applied to Conformance Checking.

**Chapter 4:** here we present our method and how we addressed the challenges that we encountered in order to develop an effective algorithm.

**Chapter 5:** in this chapter we present the results obtained by evaluating our method's performance using a synthetic dataset. The results are also compared with the existing tools.

**Chapter 6:** finally, we summarize our observations to evaluate the performance of our method in different situations, and we will draw our conclusions.

# Chapter 2

# Process Mining Fundamentals

*In this chapter we aim to present the basic notions of Process Mining, including his historical developments, objectives and tools useful in the next chapters*

## 2.1 Process Mining

Process Mining is a research field positioned between data science and business process modeling and analysis. Its goal is to extract meaningful insights from data collected by information systems, such as identifying bottleneck, providing insights, anticipating problems, recommending countermeasure and streamline processes. The starting point of Process Mining is the event log. We assume that for all processes is it possible to record sequentially the activities performed. For example, in the process of booking a hotel room, each action, like paying with a credit card, can be stored in a dataset to represent the process executions. These recordings are stored in event logs, acting as a register of past executions of the process. Each operation is recorded as an event, representing one execution of an activity, often with additional information such as the resource (person or device performing that action) or the timestamp. Event logs serves as the basis for three main types of Process Mining:

- **Discovery:** Discovery techniques automatically generate a model from an event log without any a-priori information. For example, the $\alpha$-*algorithm* produces a model that precisely illustrates how procedures are executed. If the data includes information about resources, resource-related models can also be discovered.

- **Conformance Checking:** This approach verifies if the real process executions conform to the discovered model. It involves comparing the event log with the process model of the same process to detect deviations and measure their severity. Chapter 3 will focus on this topic.

- **Enhancement:** Enhancement aims to improve or extend an existing process model using information from the event log. For instance, by leveraging timestamp information in the event log, the model can be extended to identify bottlenecks, frequencies and throughput times.

We can see the workflow of Process Mining analysis in Figure 2.1.

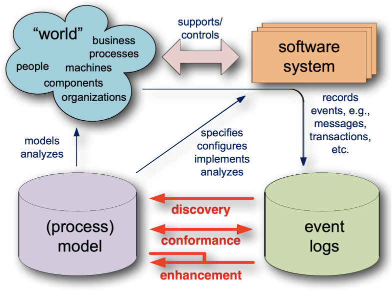


**Figure 2.1:** Positioning of the three main types of Process Mining: Discovery, Conformance Checking, Enhancement.

## 2.2 Notations and Definitions

In this chapter, we present the preliminary definitions that will be used later in this thesis. Some basic mathematic definitions will be recalled. Then, we describe the basic concepts related to Process Mining as process models and event logs.

### 2.2.1 Basic Notations

The following definitions are basic Process Mining notations used to be more mathematically precise in the more complex concepts like Petri net and Event Log[1].

**Definition 2.2.1** (Multisets)**.** Let $X$ be a set, a *multiset* of $X$ is a mapping $M : X \rightarrow N$. $B(X)$ denotes the set of all *multisets* over $X$.
Let $M$ and $M'$ be *multisets over* $X$. $M$ contains $M'$, denoted $M \geq M'$, if and only if $\forall_{x \in X} M(x) \geq M'(x)$.
The union of $M$ and $M'$ is denoted $M + M'$, and is defined by $\forall_{x \in X}(M + M')(x) = M(x) + M'(x)$.
The difference between $M$ and $M'$ is denoted $M - M'$ and is defined by $\forall_{x \in X}(M - M')(x) = (M(x) - M'(x)) \max 0$.
Note that $(M - M') + M' = M$ only holds if $M \geq M'$. For sets $X$ and $X'$ such that $X' \subseteq X$, we consider every set $X'$ to be an element of $B(X)$, where $\forall_{x \in X'} X'(x) = 1$ and $\forall_{x \in X \setminus X'} X'(x) = 0$.

**Definition 2.2.2** (Projection on sequences and multisets)**.** Let $X$ be a set, let $X' \subseteq X$ be a subset of $X$, let $\sigma \in X^*$ be a sequence over $X$, and let $M \in B(X)$ be a multiset over $X$. With $\sigma \restriction_{X'}$ we denote the projection of $\sigma$ on $X'$, e.g. $\langle x, x, y, y, y, z \rangle \restriction_{\{x,z\}} = \langle x, x, z \rangle$. with $M \restriction_{X'}$ we denote the projection of $M$ on $X'$, e.g. $[x^2, y^3, z] \restriction_{\{x,z\}} = [x^2, z]$.

**Definition 2.2.3** (Function domains and ranges)**.** Let $f \in X \nrightarrow X'$ be a *partial* function. With $dom(f) \subseteq X$ we denote the set of elements from $X$ that are mapped onto some value in $X'$ by $f$. With $rng(f) \subseteq X'$ we denote the set of elements in $X'$ that are mapped onto by some value in $X$, i.e., $rng(f) = \{f(x) | x \in dom(f)\}$.

### 2.2.2 Petri Net

In Process Mining, processes are represented using process models. There are many modelling languages to depict process: Business Process Modelling Notation (BPMN),

[1]The definitions provided has been taken from [1]

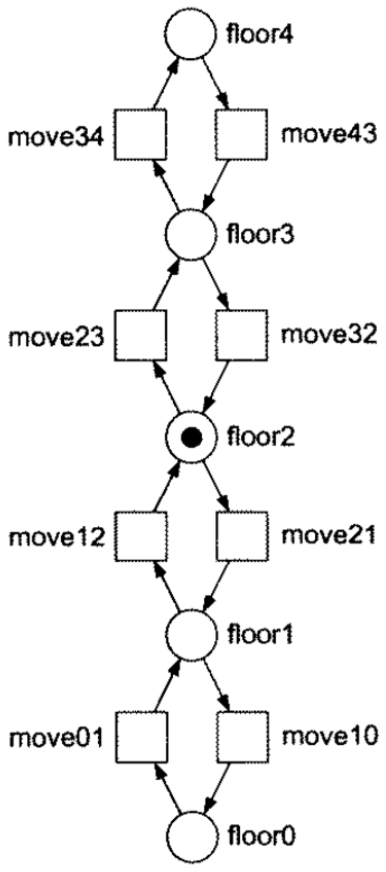


**Figure 2.2:** The elevator behavior modeled as Petri net. Source: [2]

Event-Driven Process Chains (EPCs), Unified Modeling Language (UML), Yet Another Workflow Language (YAWL) are some examples.

The most often-used process modelling notation is *Petri nets*. Petri nets are very simple and well-studied models, so in this thesis we use them to illustrate examples.

**Definition 2.2.4** (Petri net)**.** A *Petri net* is a tuple $N = (P, T, F)$ with $P$ the set of places, $T$ the set of transitions, $P \cap T = \varnothing$ and $F = (P{\times}T) \cup (T{\times}P)$ the set of arcs, which is sometimes referred to as the flow relation.

Graphically, places are visualized by circles, whereas transitions are visualized by squares. We modelled the behavior of an elevator using the Petri net figured in 2.2. The elevator moves to five floors and can stop in any of them. We denote this model as $N_1 = (P_1, T_1, F_1)$, and we represent each floor with a place, so the set of places is $P_1 = \{floor0, floor1, floor2, floor3, floor4\}$. The set of transitions is $T_1 = \{move01, move10, ..., move34\}$ which models the actions of the elevator: for example it can move from floor0 to floor1. The set of arcs is $F_1 = \{(floor0, move01), .., (move34, floor4)\}$. Arcs represent the direction of elevator's moves.

The state of a Petri net is called a *marking*, and corresponds to a multiset of places. A marking is typically visualized by putting as many so-called tokens (black dots) at a place as the place occurs in the marking. For example, a possible marking of the net $N_1$ is $[floor2]$ which is visualized by one token at place *floor2*.

**Definition 2.2.5** (Marking)**.** Let $N = (P, T, F)$ be a Petri net. A marking $M$ is a multiset of places, i.e. $M \in B(P)$.

Let $N = (P, T, F)$ be a Petri net. For a node $n \in P \cup T$ (a place or a transition), $\cdot n = \{n' | (n', n) \in F\}$ denotes the set of input nodes and $n \cdot = \{n' | (n, n') \in F\}$ denotes the set of output nodes. Transitions can change the distribution of tokens over the places. For example, transition *move23* can take the token from *floor2* and put a new token on place *floor3*. This causes a change of distribution of tokens that corresponds to the state transition in which the elevator moves from the second to the third floor. We call this the *firing* of transition *move23*. The firing of transition is subjected by rules, that will follow more precisely.

A transition $t \in T$ is enabled by a marking $M$ if and only if each of its input places $\cdot t$ contains at least one token in $M$, that is, if and only if $M \geq \cdot t$. An enabled transition may fire by removing one token from each of the input places $\cdot t$ and producing one token at each of the output places $t \cdot$. The firing of an enabled transition $t$ in marking $M$ is denoted as $(N, M)[t\rangle(N, M')$, where $M' = (M - \cdot t) + t \cdot$ is the resulting new marking. A marking $M'$ is reachable from a marking $M$ if and only if there is a sequence of transitions $\sigma = \langle t^1, t^2, ..., t^n\rangle \in T^*$such that $\forall_{0 \leq i < n} (N, M^i)[t^{i+1}\rangle(N, M^{i+1})$ with $M^0 = M$ and $M^n = M'$.

In order to make more readable activities performed in processes, we use *Labeled Petri net*. It follows the definition.

**Definition 2.2.6** (Labeled Petri net)**.** A labeled Petri net $N = (P, T, F, l)$ is a Petri net $(P, T, F)$ with labeling function $l \in T \nrightarrow \mathcal{U}_A$ where $U_A$ is some universe of activity labels. Let $\sigma_v = \langle a1, a2, ..., an\rangle \in \mathcal{U}_A^*$ be a sequence of activities. $(N, M)[\sigma_v \triangleleft (N, M')$ if and only if there is a sequence $\sigma \in T^*$ such that $(N, M)[\sigma\rangle(N, M')$ and $l(\sigma) = \sigma_v$.

The labeling function is used to map each transition to the corresponding activity in the process. A transition $t$ is called *invisible* if and only if it is not mapped to any activity label by the labeling function, that is, if and only if $t \notin dom(l)$. Otherwise, transition $t$ is *visible* and corresponds to an observable activity $a = l(t)$.

In Process Mining we are more interested in process with an initial state and a well-defined final state. We say that a firing sequence is complete when it starts from a marking containing an initial place and ends in a marking only containing the final places. For example, in Figure 2.2 if we assume as initial place *floor0* and final place *floor4*, a complete firing sequence can be $\langle move01, move12, move23, move34\rangle$. The notion of System Net is provided to include the concepts of initial and final state.

**Definition 2.2.7** (System net)**.** System net is a triplet $SN = (N, I, O)$ where $N = (P, T, F, l)$ is a labeled Petri net, $I \in B(P)$ is the initial marking and $O \in B(P)$ is the final marking. $\mathcal{U}_{SN}$ is the universe of system nets.

More notations about System Net are given:

| Case id | Event id | Properties | | | | |
|---|---|---|---|---|---|---|
| | | Timestamp | Activity | Resource | Cost | ... |
| 1 | 35654423 | 30-12-2010:11.02 | register request | Pete | 50 | ... |
| | 35654424 | 31-12-2010:10.06 | examine thoroughly | Sue | 400 | ... |
| | 35654425 | 05-01-2011:15.12 | check ticket | Mike | 100 | ... |
| | 35654426 | 06-01-2011:11.18 | decide | Sara | 200 | ... |
| | 35654427 | 07-01-2011:14.24 | reject request | Pete | 200 | ... |
| 2 | 35654483 | 30-12-2010:11.32 | register request | Mike | 50 | ... |
| | 35654485 | 30-12-2010:12.12 | check ticket | Mike | 100 | ... |
| | 35654487 | 30-12-2010:14.16 | examine casually | Pete | 400 | ... |
| | 35654488 | 05-01-2011:11.22 | decide | Sara | 200 | ... |
| | 35654489 | 08-01-2011:12.05 | pay compensation | Ellen | 200 | ... |
| 3 | 35654521 | 30-12-2010:14.32 | register request | Pete | 50 | ... |
| | 35654522 | 30-12-2010:15.06 | examine casually | Mike | 400 | ... |
| | 35654524 | 30-12-2010:16.34 | check ticket | Ellen | 100 | ... |
| | 35654525 | 06-01-2011:09.18 | decide | Sara | 200 | ... |
| | 35654526 | 06-01-2011:12.18 | reinitiate request | Sara | 200 | ... |

**Figure 2.3:** An example of an event log. Source: [3]

**Definition 2.2.8** (System net notations)**.** Let $SN = (N, I, O) \in \mathcal{U}_{SN}$ be a system net with $N = (P, T, F, l)$.

* $T_v(SN) = dom(l)$ is the set of visible transitions in $SN$.
* $A_v(SN) = rng(l)$ is the set of corresponding observable activities in $SN$.
* $T_v^u(SN) = \{t \in T_v(SN) | \forall_{t' \in T_v(SN)} l(t) = l(t') \Rightarrow t = t'\}$ is the set of unique visible transitions in $SN$ (such that no other transition has the same visible label).
* $A_v^u(SN) = \{l(t) | t \in T_v^u(SN)\}$ is the set of corresponding unique observable activities in $SN$.

### 2.2.3 Event Logs

As already mentioned, the starting point of Process Mining is the event log. In event log is stored the amount of data representing the execution of a particular process. An event log is a multiset of *traces*. Each trace is a complete execution of activities in the process. In Figure 2.3 we can see an example of an event log about a process related to the handling of requests for compensation. Here we refer to each trace as *case*. Each trace consists in a list of ordered events which represent an execution of the process. Events represent the activities executed, that can be linked to the transitions of the model in a second moment. In our example, events are enriched with other additional information, called *attributes*, such as resources, timestamps or additional data elements recorded with the event log. In *case id: 1* the first event is related to the activity *register request*, performed by resource *Pete* at *11:02* of the *12th December 2010*, and it costs *20*. The additional information is more used in the third type of Process Mining, the Enhancement. In this thesis we will abstract from such information and limit conformance to solely the control flow aspect.

The followings are the formal definitions about the concepts described above.

**Definition 2.2.9** (Traces)**.** Let $SN = (N, I, O) \in \mathcal{U}_{SN}$ be a system net. $\phi(SN) = \{\sigma_v | (N, I)[\sigma_v \triangleleft (N, O)\}$ is the set of visible traces starting in marking I and ending in marking O. $\phi_f(SN) = \{\sigma | (N, I)[\sigma\rangle(N, O)\}$ is the corresponding set of complete firing sequences.

**Definition 2.2.10** (Event Log)**.** An event log $L \in B(A^*)$ is a multiset of traces, where $A$ is a set of activities.

The standard about storing events logs is *XES (eXtensible Event Stream)* . In the next section we will describe this format in detail.

## 2.3 Tools for Process Mining

### 2.3.1 ProM

ProM (short for Process Mining framework) is an Open Source framework for Process Mining algorithms. It has become the de facto standard Process Mining platform in the academic world by establishing an active, recognized community of contributors and users. This platform is extensible and platform independent as it is implemented in Java.

In 2002, several researchers were building simple prototypes to experiment with process discovery techniques. It became apparent that building a dedicated Process Mining tool for every newly conceived Process discovery method was not practical. This realization led to the development of the *ProM* framework, a "plug-able" environment for Process Mining algorithms, with the goal of creating a common basis for all kind of techniques. The first version of *ProM* framework was released in 2004, using MXML as input format for representing event logs. The current version is *ProM* 6, released in 2010, based on a new architecture and utilizing the XES format.

One significant upgrade of ProM 6 is its ability to distribute the execution of plug-ins over multiple computers. This feature enhances performance and allows ProM to be offered as a service. The ProM environment is composed by many plug-ins, each providing different functionalities. Currently, there are over 1500 plug-ins available. Developers can create numerous plug-ins, and loading them in ProM as users can utilize them. Plug-ins are distributed over so-called *packages* and can be chained into composite plug-ins. Packages contain related sets of plug-ins. ProM 6 includes a package manager for adding, removing, and updating packages. Users should only load packages that are relevant to the tasks they want to perform.

### 2.3.2 XES

As mentioned earlier, event logs are the starting point of Process Mining analysis, and various formats exist to store this event data. Until 2010, the de facto standard was *MXML (Mining eXtensible Markup Language)*, which emerged in 2003 and was adopted by the early version of ProM tool. While this approach worked reasonably well, it presented various extensibility limitations. This led to the development of the *XES (eXtensible Event Stream)* format. XES was adopted in 2010 by the *IEEE Task Force on Process Mining* and has since become the de facto exchange format of event logs. The IEEE Standards Organization is currently evaluating XES with the aim of turning it into an official IEEE standard [3]. Its success is attribute to fact that it is

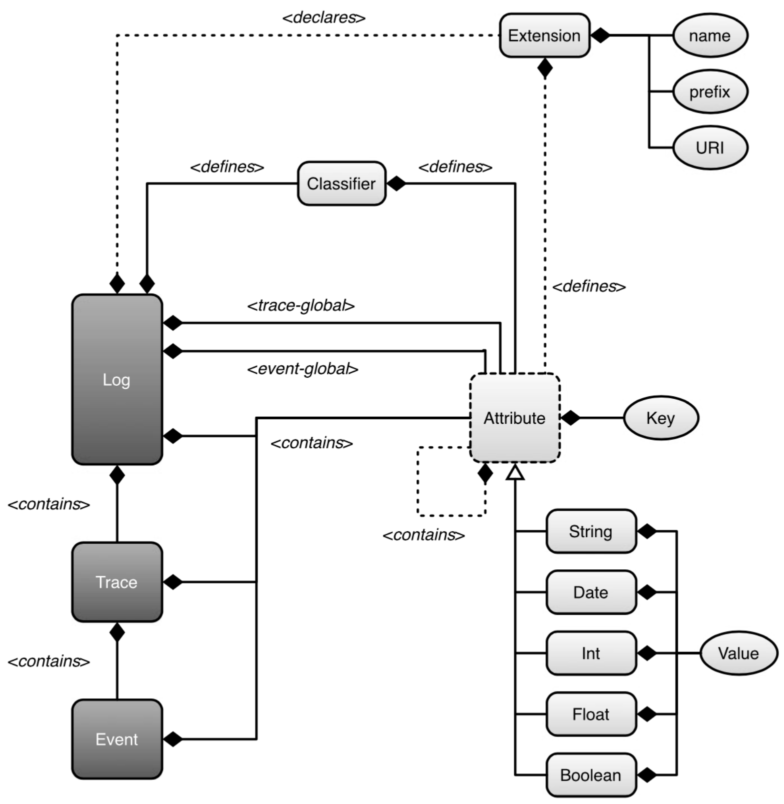


**Figure 2.4:** Meta-model of XES expressed using UML class diagrams. Source: [3]

less restrictive than its predecessor and is truly extensible.

The XES meta-model, expressed in terms of a UML class diagram, is pictured in Figure 2.4. An XES document contains one log formed by any number of traces. Each trace describes a sequential list of events. The log, its traces, and its events may have any number of attributes, which may also be nested. An event can have any number of attributes, and there is no fixed set of mandatory attributes. However, to provide semantics for such attributes, the log refers to so-called *extensions*. For example, the *Time extension* defines a timestamp attribute of type dataTime. It is possible to define domain-specific extensions to be more expressive for the user.

Another important feature of XES is that it supports the *classifier* concept. The classifier is a function that maps the attributes of an event onto a label used in the resulting process model. Each classifier is specified by a list of attributes. Any two events that have the identical values with respect to these attributes are considered to be equal for that classifier.

Every tool we use in this thesis fully supports the XES standard. Moreover, this is the format we use to create the dataset for experiments.

# Chapter 3

# Conformance Checking

*In this chapter we present the area of Conformance Checking and techniques to approach the problem. Then we show how to apply decomposition-based approaches to Conformace Checking. This includes the procedures to split the model in a valid way and the existing decomposition-based algorithms*

## 3.1 Preliminaries

Conformance Checking is one of the three main areas of Process Mining. This thesis is specifically focused on this area.
Conformance Checking techniques establish connections between events in the event log and activities in the process model, comparing both to identify commonalities and discrepancies. In essence, these techniques analyze observed process executions in comparison to the modeled behavior, providing global conformance measures to quantify the overall conformance of the model and log. Conformance Checking can be related to the standard *ISO 9000:2008*, which requires organization to model their operational processes. This standard aims to mitigate risks and regulate processes in modern companies, making Process Mining a valuable tool, especially in *business alignment* and *auditing* fields.

The objective of **business alignment** is to effective alignment of information systems with real business processes. Process Mining contributes to improving this alignment by analyzing actual business processes, diagnosing discrepancies, and providing insights on enhancing the support offered by information systems.

**Auditing** involves evaluating organizations and their processes to verify the validity and reliability of information. This is done to check whether business processes are executed within certain boundaries set by managers, governments, and other stakeholders.

Conformance Checking techniques play a crucial role in both fields by helping detect fraud, malpractice, risks, and inefficiencies. They contribute to the evolution of a new form of auditing that evaluates all events in a business process while the process is still ongoing.

As already written above, Conformance Checking is useful to detect discrepancies. We can divide these deviations in desirable and undesirable. When the procedures encoded in the process models are inefficient, we can see *desirable deviations*. *Undesiderable deviations* appear when the procedures are established and solid, but process actors do not comply them.
Basically, Conformance Checking can establish if the model used to represent a certain process is a good model. We use four quality dimensions to determine the quality of a process mining model: *fitness, simplicity, precision, and generalization* [3]. To create a good model we should balance all these aspects:

* A model with good **fitness** allows for the behavior seen in the event log. A model has a perfect fitness if all traces in the log can be replayed by the model from beginning to end. There are different ways to define fitness, we will see them in detail in the following sections.

* The **simplicity** dimension refers to the fact that the best model is the simplest model that can explain the behavior seen in the log. The complexity of the model could be defined by the number of nodes and arcs in the underlying graph.

* A model is **precise** if it does not allow for "too much" behavior. A model that is not precise is under fitting. Under fitting is the problem that the model over-generalizes the example behavior in the log.

* A model should **generalize** and not restrict behavior to the examples seen in the log. A model that does not generalize is over fitting. Over fitting is the problem that a very specific model is generated whereas it is obvious that the log only holds example behavior.

Another important use of Conformance Checking is the evaluation of discovery algorithms. In the first area of Process Mining, discovery, we *mine* the event log to create model. There are several algorithms to produce process models. It is possible to evaluate each model produced by different algorithms using Conformance Checking. In this way, we can detect the best model to continue the analysis.

We provide the two most famous conformance checking algorithms: *token replay* and *alignment-based conformance checking*. In this thesis, we use the fitness measure to quantify the quality of the model. In both algorithm descriptions, an accurate definition of fitness metric is provided. We preferred the **alignment-based** approach as it can pinpoint exactly when and where deviations occurred.

### 3.1.1 Token-Replay Conformance Checking

*Replay* means that we use process model and event log as inputs then we compare the behaviors observed in the log with the ones performed by the model. *Token replay* approach consist in taking a trace from the log and try to replay it in the model [3]. To understand better how the algorithm works, we describe an example, figured in Image 3.1 where the various stages of replay are represented. When we perform token replay, four counters are taking in consideration:

* produced tokens (p);

* consumed tokens (c);

- $*$ missing tokens (m);
- $*$ remaining tokens (r).

Initially, $p = c = 0$ and all places are empty. Then the environment produces a token for place *start*. Consequently, counter $p$ is incremented by 1, as one token is being produced. Let consider reproducing the trace $\sigma_1 = \langle a, d, c, e, h\rangle$.

The first transition we have to fire is $a$. This is possible. Since $a$ consumes one token and produces two tokens, we update $c$ by incrementing by 1 and $p$ by incrementing by 2. The resulting configuration is $p = 3$, $c = 1$, $m = 0$, $r = 0$. At this point we try to replay second event $d$, but this is not possible as the transition is not enabled. We need to add a token in *place2* to fire $d$, so we record this missing token by incrementing $m$ by 1 and by updating $p$ and $c$. We are in the third stage of the image. The following activities *c, e, h* can be easily performed by the model without adding any new token, so we skip to the last configuration showed in the figure. After replaying the last event, we have $p = 6$, $c = 5$, $m = 1$, and $r = 0$. In the final state $[p2, end]$ the environment consumes the token from place *end*. However, a token is left in place *p2*, so counter $r$ is incrementing by 1. Hence the final result is $p = c = 6$ and $m = r = 1$.

The final configuration of counters provides information about non-conformance. There was a situation in which $d$ occurred but could not happen according to the model
(m = 1) and there was a situation in which $d$ was supposed to happen but did not occur according to the log (r = 1).

Moreover, using the counters we can compute fitness according to the following formula:

$$fitness(\sigma, N) = \frac{1}{2}\left(1 - \frac{m}{c}\right) + \frac{1}{2}\left(1 - \frac{r}{p}\right)$$

In our example, $fitness(\sigma_1, N) = \frac{1}{2}\left(1 - \frac{1}{6}\right) + \frac{1}{2}\left(1 - \frac{1}{6}\right) = 0,833$.

It is possible to compute the fitness of an event log using replay token approach with the following more generic formula:

$$fitness(L, N) = \frac{1}{2}\left(1 - \frac{\sum_{\sigma \in L} L(\sigma) * m_{N,\sigma}}{sum_{\sigma \in L} L(\sigma) * c_{N,\sigma}}\right) + \frac{1}{2}\left(1 - \frac{sum_{\sigma \in L} L(\sigma) * r_{N,\sigma}}{sum_{\sigma \in L} L(\sigma) * p_{N,\sigma}}\right)$$

The value of fitness is between 0 (very poor fitness) and 1 (perfect fitness).

### 3.1.2 Alignment-Based Conformance Checking

Token replay is a really easy algorithm to understand and can be implemented efficiently. However, this approach has also some limitations. First, it is Petri net specific, and can only be applied to other representations after their conversions. Moreover, if a case does not fit, the method does not create a corresponding path through the model. We would like to map observed behavior onto modeled behavior to provide better diagnostics and to relate also non-fitting cases to the model. In other words, we would like to pinpoint deviations in terms of activities not executed at the right time. The concept of *alignments* is introduced to overcome these drawbacks. The main idea is to find an execution of the model that is close as possible to the observed trace. We

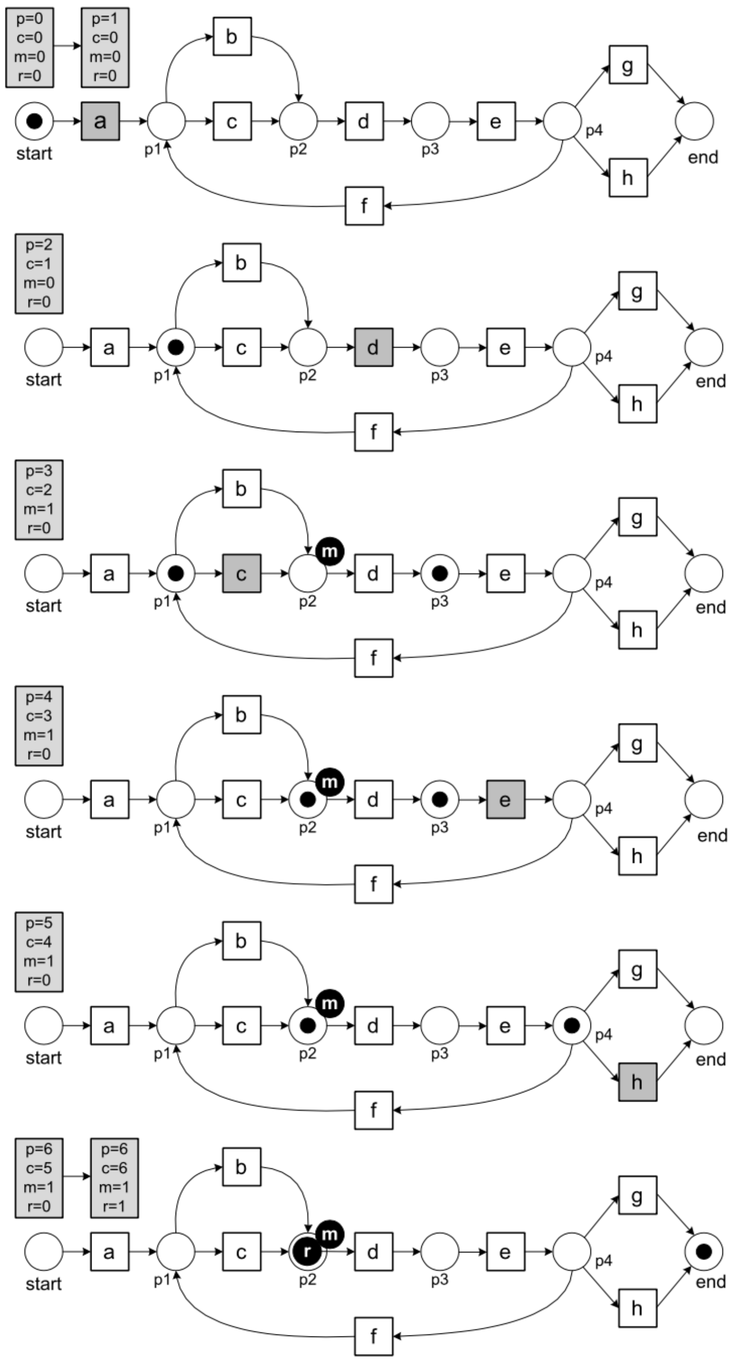


**Figure 3.1:** Replaying of trace $\sigma_1 = \langle a, d, c, e, h\rangle$ on the net $N$. The various stages of replay are represented with the according four counters: p (produced tokens), c (consumed tokens), m (missing tokens), and r (remaining tokens). Source: [3]

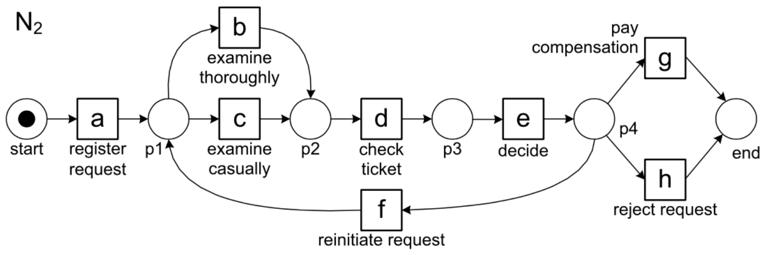


**Figure 3.2:** Petri Net $N_2$. Source: [3]

explain this concept by making of an example. Let consider the easy Petri net $N_2$ in Figure 3.2 and trace $\sigma_2 = \langle a, d, b, e, h \rangle$. We compute the following alignment:

$$\gamma_1 = \begin{array}{|c|c|c|c|c|c|} a & \gg & d & b & e & h \\ \hline a & c & d & \gg & e & h \end{array}$$

The first row corresponds to the trace in the event log whereas the second row corresponds to a firing sequence in the model. Our intention is to try to mimic an activity in the model by using an activity in the trace. If an activity in the model cannot be mimicked by an activity in the log, then the symbol $\gg$ (no step) appears in the top row. Similarly, if an activity in the log cannot be mimicked by an activity in the model, then a $\gg$ (no step) appears in the bottom row. In our example, the trace starts with activity $a$. Our model also begins with transition $a$. Hence, we can perform this move both on log and model size. Instead in second column, the model fires transition $c$ before $d$, exposing an incoherence with the log, as it is not possible for the model to make a $d$ move immediately after $a$. In forth column, it is possible to see a *log move*: the log activity $b$ is performed after $d$, but the model could not mimic this behavior. It is also possible to perform *invisible transitions*, in this case we use the symbol $\tau$.

To be more precise, we define a *move* as a pair $(a, m)$ where the first element $a$ refers to the activity in the log and the second element $m$ refers to the transition in the net. The following is the definition of possible moves in an alignment:

**Definition 3.1.1** (Legal moves)**.** Let $L \in B(A^*)$ be an event log and let $SN = (N, I, O) \in \mathcal{U}_{SN}$ be a system net with $N = (P, T, F, l)$. $A_LM = \{(a, (a, t)) | a \in A \wedge t \in T \wedge l(t) = a\} \cup \{(\gg, (a, t)) | a \in A \wedge t \in T \wedge l(t) = a\} \cup \{(\gg, (\tau, t)) | t \in T \wedge t \backslash \in dom(l)\} \cup \{(a, \gg) | a \in A\}$ is the set of legal moves. The function $\alpha \in A_{LM} \rightarrow A \cup \{\tau\}$ provides the activity (possibly $\tau$) associated with a move: for all $t \in T$ and $a \in A$

* $\alpha(a, (a, t)) = a$ : synchronous move;
* $\alpha(\gg, (a, t)) = a$ : model move;
* $\alpha(\gg, (\tau, t)) = \tau$ : invisible move;
* $\alpha(a, \gg) = a$ : log move.

An alignment is a sequence of moves. This means that, after removing all $\gg$ symbols, the first row corresponds to a log trace and the bottom row corresponds to a firing sequence in the net from initial marking to final marking.

**Definition 3.1.2** (Alignment)**.** Let $L \in B(A^*)$ be an event log with $A \subseteq \mathcal{U}_A$, let $\sigma_L \in L$ be a log trace and $\sigma_M \in \phi_f(SN)$ a complete firing sequence of system net SN. An alignment of $\sigma_L$ and $\sigma_M$ is a sequence $\gamma \in A^*_{LM}$ such that the projection on the first element (ignoring any $\gg$) yields $\sigma_L$ and the projection on the last element (ignoring any $\gg$) yields $\sigma_M$.

Given a log trace and a process model, there may be many (if not infinitely many) alignments. Returning to our example, we can provide three more alignments:

**Alignments 3.1.1.**

$$\gamma_2 = \begin{array}{|c|c|c|c|c|c|} a & \gg & d & b & e & h \\ \hline a & b & d & \gg & e & h \end{array} \qquad \gamma_3 = \begin{array}{|c|c|c|c|c|c|} a & d & b & \gg & e & h \\ \hline a & \gg & b & d & e & h \end{array}$$

$$\gamma_4 = \begin{array}{|c|c|c|c|c|c|c|c|c|} a & \gg & d & \gg & \gg & b & \gg & e & h \\ \hline a & c & d & e & f & b & d & e & h \end{array}$$

Intuitively $\gamma_4$ is the worst alignment as it has a bigger number of misalignment moves. To select the most appropriate alignment, we associate *costs* to undesirable moves and select the alignment with the lowest cost.

**Definition 3.1.3** (Cost of alignment)**.** The cost function $\delta \in A_{LM} \rightarrow Q$ assigns costs to legal moves. Moves where the log and the model agree have no costs, i.e., $\delta(a,(a,t)) = 0$ for all $a \in A$. A move in the model also has no costs if the transition is invisible, i.e., $\delta(\gg,(\tau,t)) = 0$ if $t \notin dom(l)$. A move in the model has a cost of $\delta(\gg,(a,t)) > 0$ if $l(t) = a$ and $a \in A$. Similarly, a move in the log has a cost of $\delta(a,\gg) > 0$. The cost of an alignment $\gamma \in A^*$ is the sum of all costs: $\delta(\gamma) = \sum_{(a,m)\in\gamma} \delta(a,m)$.

For simplicity, in this thesis we fixed a standard cost function $\delta_1$ that assigns cost 1 to all visible model moves and log moves. The costs of our examples are: $\gamma_1 = 2, \gamma_2 = 2, \gamma_3 = 2, \gamma_4 = 4$.
Using the cost function we can define optimal alignment:

**Definition 3.1.4** (Optimal alignment)**.** Let $L \in B(A^*)$ be an event log with $A \subseteq \mathcal{U}_A$ and let $SN \in \mathcal{U}_S N$ be a system net with $\phi(SN) \neq \emptyset$.

* For $\sigma_L \in L$, an alignment $\gamma$ between $\sigma_L$ and a complete firing sequence of the system net $\sigma_M \in \phi_f(SN)$ is optimal if the associated misalignment costs are lower or equal to the costs of any other possible alignment $\gamma'$.
* $\lambda(\sigma_L, SN) \in A^* \rightarrow A^*_{LM}$ is a deterministic mapping that assigns any log trace $\sigma_L$ to an optimal alignment.

The optimal alignments for our example is $\gamma_1, \gamma_2, \gamma_3$. This shows optimal alignments do not need to be unique.

Using these definitions we can illustrate how to compute fitness using alignments. First we have to introduce the concept of *worst alignment*. The worst alignment is an alignment with no synchronous moves but only log moves and model moves. It is constructed as the sequence of log moves for all events, followed by the sequence of model moves for each transition in the shortest path from the initial to the final marking. For example:

**Alignments 3.1.2** (Worst alignment)**.**

$$\gamma_5 = \begin{array}{|c|c|c|c|c|c|c|c|c|c|} a & d & b & e & h & \gg & \gg & \gg & \gg & \gg \\ \hline \gg & \gg & \gg & \gg & \gg & a & c & d & e & h \end{array}$$

The cost of our worst alignment is 10.
The fitness of a single trace $\sigma_L$ is computed as follows:

$$fitness(\sigma_L, SN, \delta) = 1 - \frac{\delta(\lambda^N_{opt}(\sigma))}{\delta(\lambda^N_{worst}(\sigma))}$$

For our example the fitness is:

$$fitness(\sigma_1, SN, \delta_s td) = 1 - \frac{2}{(10)} = 0,8$$

It is possible to compute the fitness of an event log using alignments approach with the following more generic formula:

**Definition 3.1.5** (Fitness)**.**

$$fitness(\sigma_L, SN, \delta) = 1 - \frac{\sum_{\sigma \in L} L(\sigma) \cdot \delta(\lambda^N_{opt}(\sigma))}{\sum_{\sigma \in L} L(\sigma) \cdot \delta(\lambda^N_{worst}(\sigma))}$$

The fitness metric has the same meaning as in token replay approach. The fitness metric computes a value between 0 and 1. A trace that perfectly fits the system net would yield a fitness value of 1 and a trace that does not fit the system net at all would yield a fitness value of 0.

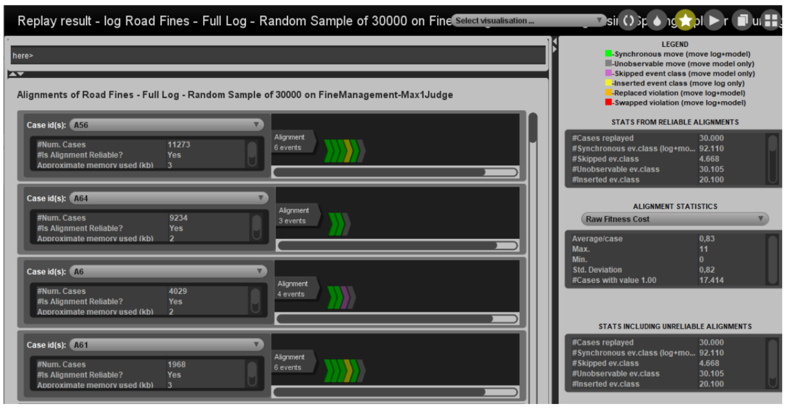
In conclusion, alignments are powerful because they can explain where and which deviations occur. For example, we can indicate that a specific activity is often skipped or that some other activity occurs at times it is not supposed to happen. Moreover, observed behavior is related to modeled behavior in a precise manner. Another advantage respect to token replay approach is that alignments can be defined for any process notation, including Petri nets having duplicate and silent activities. In the following subsection, we present the algorithm implemented to compute alignments.

### 3.1.3 Conformance Checking Algorithm

To compute alignments is used the plug-in implemented in ProM *"Replay a log on Petri Net for conformance analysis"*. The plug-in takes a Petri net model and the event-log as inputs, then it produces alignments as output (see Figure 3.3 for an example).

The problem of finding alignments can be expressed the following way: *given a synchronous product Petri net and a cost function, provide the cheapest firing sequence from the initial marking to the final marking.*

To construct the *synchronous product* we start from the original Petri net model. Then we add the trace as a sequential Petri net and the synchronous transitions. Synchronous moves are created by pairing each activity in the trace to a transition in the model that corresponds to the same activity. In Figure 3.4 the original net is colored gray, the net constructed by the trace is yellow and the synchronous transitions



Replay result - log Road Fines - Full Log - Random Sample of 30000 on Fine

Select visualisation ...

here>

Alignments of Road Fines - Full Log - Random Sample of 30000 on FineManagement-Max1Judge

Case id(s): A56

| | |
|---|---|
| #Num. Cases | 11273 |
| #Is Alignment Reliable? | Yes |
| Approximate memory used (kb) | 3 |

Alignment
6 events

Case id(s): A64

| | |
|---|---|
| #Num. Cases | 9234 |
| #Is Alignment Reliable? | Yes |
| Approximate memory used (kb) | 2 |

Alignment
3 events

Case id(s): A6

| | |
|---|---|
| #Num. Cases | 4029 |
| #Is Alignment Reliable? | Yes |
| Approximate memory used (kb) | 2 |

Alignment
4 events

Case id(s): A61

| | |
|---|---|
| #Num. Cases | 1968 |
| #Is Alignment Reliable? | Yes |
| Approximate memory used (kb) | 3 |

Alignment
6 events

LEGEND

- 🟩-Synchronous move (move log+model)
- ⬜-Unobservable move (move model only)
- 🟪-Skipped event class (move model only)
- 🟨-Inserted event class (move log only)
- 🟧-Replaced violation (move log+model)
- 🟥-Swapped violation (move log+model)

STATS FROM RELIABLE ALIGNMENTS

| | |
|---|---|
| #Cases replayed | 30.000 |
| #Synchronous ev.class (log+mo... | 92.110 |
| #Skipped ev.class | 4.668 |
| #Unobservable ev.class | 30.105 |
| #Inserted ev.class | 20.100 |

ALIGNMENT STATISTICS

Raw Fitness Cost

| | |
|---|---|
| Average/case | 0,83 |
| Max. | 11 |
| Min. | 0 |
| Std. Deviation | 0,82 |
| #Cases with value 1.00 | 17.414 |

STATS INCLUDING UNRELIABLE ALIGNMENTS

| | |
|---|---|
| #Cases replayed | 30.000 |
| #Synchronous ev.class (log+mo... | 92.110 |
| #Skipped ev.class | 4.668 |
| #Unobservable ev.class | 30.105 |
| #Inserted ev.class | 20.100 |

**Figure 3.3:** Visualization of the result of *"Replay a log on Petri Net for conformance analysis"* plug-in of ProM software. We can see for each line, the alignment corresponding to a trace.

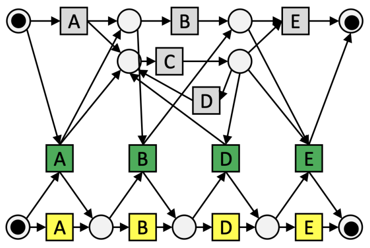
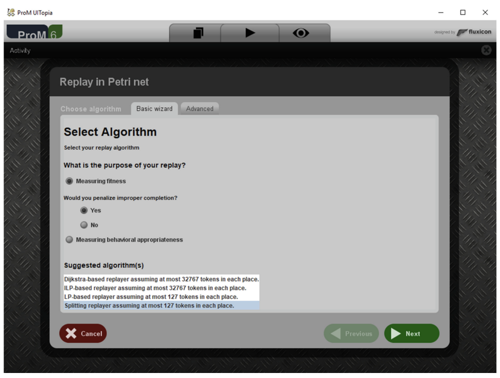


**Figure 3.4:** *Synchronous product* Petri net. In gray is pictured the original net; in yellow the transitions corresponding to activities in the trace, and in green the synchronous transitions.

are green. Starting from this new net we need to find the shortest execution, where the length of the path is defined by the cost function. It is clear that this is a *search problem*. The search space is the statespace of the synchronous product model. Each node is a combination of a state in the model and the remaining events in the trace. Each arc is a move on model, move on log or a synchronous move.

The plug-in provides different search algorithms to find the shortest path. Some example can be using Dijikstra algorithm or A* algorithm. The most recent algorithm and more efficient [4] is the *Iterative A* technique*, in the plug-in is called "Splitting replayer assuming at most 127 tokens at each place" (Figure 3.5). The core of this algorithm is, of course, an A* search technique. In every iteration, a marking is selected for expansion. This marking $m$ is chosen such that it minimizes the cost to reach that marking from the initial marking $g(m)$ plus the estimated remaining cost $h(m)$. The algorithm stops if the final marking $m_f$ is reached. The innovation introduced is to reconsider the heuristic function $h(m)$ by exploiting knowledge of the traces being aligned. Essentially, it is used the original trace to guarantee progress in the depth of the search by splitting the marking equation (heuristic function) into a number of sub-problems which together provide a more accurate under estimation of the remaining cost. It is used the A* algorithm itself to decide when to split the



Replay in Petri net

Choose algorithm | Basic wizard | Advanced

Select Algorithm

Select your replay algorithm

What is the purpose of your replay?

[✓ Measuring fitness]

Would you penalize improper completion?

[✓ Yes] No

Measuring behavioral appropriateness

Suggested algorithm(s)

Dijkstra-based replayer assuming at most 32767 tokens in each place.
ILP-based replayer assuming at most 32767 tokens in each place.
LP-based replayer assuming at most 127 tokens in each place.
Splitting replayer assuming at most 127 tokens in each place.

Cancel | Previous | Next

**Figure 3.5:** Configuration of the parameters to build alignments in the plug-in *"Replay a log on Petri Net for conformance analysis"*. Note that the standard configuration uses the *Iterative A* technique.*

marking equation[1]. In [4] is shown that the technique leads to significant improvements in computation time.

## 3.2 Decomposition-Based Approach

In the last few years, information technology and global digitalization has grown exponentially in both academic and industry field. One of the consequences of this phenomenon is the incredible growth of data. The term *Big Data* was coined in this sense: it illustrates the spectacular growth of data and the potential economic value of such data in different industry sectors. In previous chapters, we have already presented how Process Mining can be a powerful instrument in analyzing data, with the aim of improve processes.

However, the incredible growth of event data is also posing new challenges in Process Mining world. As event logs increase, Process Mining techniques need to become more efficient and highly scalable. To overcome this problem, new methods have been invented. One idea is using a *divide and conquer approach* [6]. This approach intends to decompose the model and compute the algorithm for each sub-models. The assumption is that computing the problem into smaller entities is easier than computing the entire problem.
Decomposition approach can be used in two fields of Process Mining:

∗ to decompose *process discovery*, we split the set of activities into a collection of partly overlapping activity sets.

∗ In *Conformance Checking*, we decompose the process model into smaller partly overlapping model fragments. If the decomposition is done properly, then any trace that fits into the overall model also fits all the smaller model fragments and vice versa.

In this thesis we aim to create a new algorithm for computing Conformance Checking using this approach.

[1]We leave to the reader a more accurate description of the algorithm. It can be found in [4] and [5]

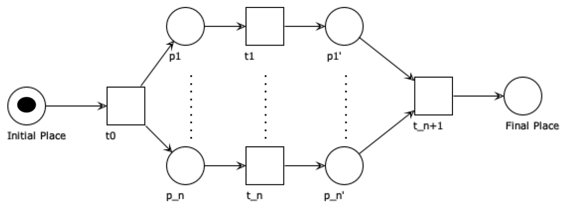


**Figure 3.6:** Petri net with $n$ parallel activities.

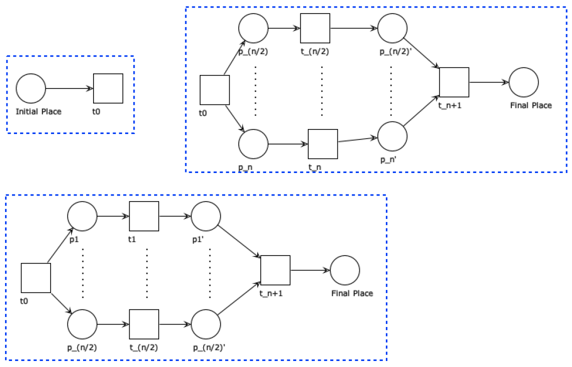


**Figure 3.7:** A possible decomposition of the net in Figure 3.6.

### 3.2.1 Decomposition in Conformance Checking

In [7] authors prove that the complexity of Conformance Checking problems are exponential with the size of the model. Using the Petri net in Figure 3.6, we show an example of how complexity can increase. The complexity of a Petri net is measured by counting how many possible traces it can accept. In the initial configuration, we have a token in *initial place* that can be fired by transition *t0*. Consequently, *t0* produces $n$ tokens in the following places and can be consumed by any of the $n$ parallel transitions. This Petri net admits $n!$ possible traces, that is of course non-linearly increase with respect to $n$.
According to the decomposition approach, we split the model into subnets and then compute the problem. In this case, we can decompose the Petri net into three subnets: the initial stage and two subnets with half of the $n$ parallel transitions for each. At this point we can execute the Conformance Checking for each subnet. The possible number of traces accepted becomes $(\frac{n}{2})!$.
We propose a more practical example. If $n = 6$, then $n! = 6! = 720$, but $\frac{6}{2}! = 6$ that is a really smaller number respect to 720 traces.

However, the decomposing approach must be aware of some aspects. First, it is important to decompose process model into model fragments in a valid way. Moreover, we have to be sure that Conformance Checking can be done locally in the subnets by using correspondingly sublogs. It is important to adapt cost function and fitness with decomposition.

The definition of *valid decomposition* is presented in [7]. If we consider a system net $SN$, it is decomposed into a collection of subnets $\{SN^1, SN^2, ..., SN^n\}$ such that

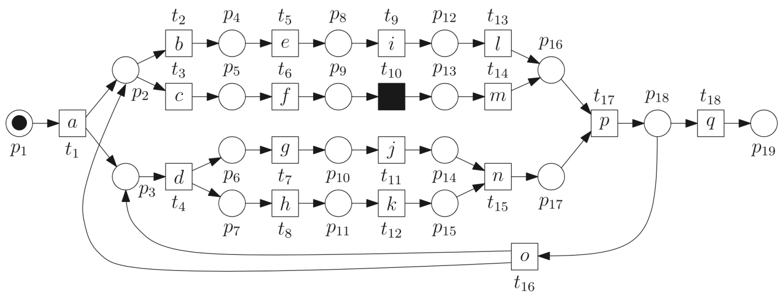


**Figure 3.8:** System net $SN_1$. Source: [1]

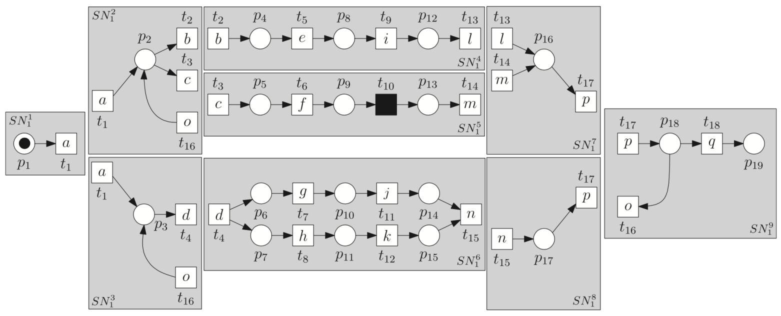


**Figure 3.9:** Valid decomposition $D_1$ of the system net $SN_1$. Source: [1]

the union of them is the original system net. A decomposition of a Petri net is valid if each place and invisible transition resides in just one subnet. Moreover, if there are multiple transitions with the same label, they should reside in the same subnet. Only unique visible transitions can be shared among different subnets. In other words, each subnet must be independent from the others.

**Definition 3.2.1** (Valid decomposition)**.** Let $SN \in \mathcal{U}_{SN}$ be a system net with labeling function $l$. $D = \{SN^1, SN^2, ..., SN^n\} \subseteq \mathcal{U}_{SN}$ is a valid decomposition if and only if:

* $SN^i = (N^i, M^i, M^i)$ is a system net with $N^i = (P^i, T^i, F^i, l^i)$ for all $1 \leq i \leq n$,
* $l^i = l \upharpoonright T^i$ for all $1 \leq i \leq n$,
* $P^i \cap P^j = \emptyset$ for $1 \leq i < j \leq n$,
* $T^i \cap T^j \subseteq T^u_{(SN)}$ for $1 \leq i < j \leq n$, and
* $SN = \bigcup_{1 \leq i \leq n} SN^i$.

$D(SN)$ is the set of all valid decompositions of $SN$.

We can see an example of valid decomposition of the Petri net $SN_1$ (Figure 3.8), in Figure 3.9. In practical words, the net has been cut along some transitions, which are called *border activities*. It is evident that these activities are shared by the subnets. This overlapping property can be used during the process of merging to pass from local results to the overall result.

**Definition 3.2.2** (Border activities)**.** Let $SN = (N, I, O) \in \mathcal{U}_{SN}$ be a system net with $N = (P, T, F, l)$. Let $D = \{SN^1, SN^2, ..., SN^n\} \in D(SN)$ be a valid decomposition of $SN$. For all $1 \leq i \leq n$, $SN^i = (N^i, I^i, O^i)$ is a subnet with $N^i = (P^i, T^i, F^i, l^i)$. $A_b(D) = \{l(t) | \exists_{1 \leq i < j \leq n} t \in T^i \cap T^j\}$ is the set of border activities of the valid decomposition $D$.
For an activity $a \in rng(l)$, $SN_b(a, D) = \{SN^i | SN^i \in D \wedge a \in A \vee (SN^i)\}$ is the set of subnets that contain $a$ as an observable activity.

A border activity can only be an activity that has a unique label. In Figure 3.9, the valid decomposition $D_1$ has the set of border activities $\{a, b, c, d, l, m, n, o, p\}$. Unique activities may appear in multiple subnets (as they can be border activities). Instead, non-unique activities will appear in exactly one subnet.
The border activity $a$ in our example appears in subnets $SN_1^1, SN_1^2, SN_1^3$, i.e. $SN_b(a, D_1) = \{SN_1^1, SN_1^2, SN_1^3\}$.
A few algorithms exist for the decomposition of Petri nets, we illustrate them in the following section.

Once we give the definition of a valid decomposition, we have to show that is possible to compute alignments for each subnet by using correspondingly projected event logs. In [7] is proved an important theorem which shows that any trace that fits the overall process model can be decomposed into smaller traces that fit the individual model fragments. Moreover, if the smaller traces fit the individual model fragments, then they can be composed into an overall trace that fits into the overall process model.

**Theorem 3.2.1** (Conformance checking can be decomposed [7])**.** *Let $L \in B(A^*)$ be an event log with $A \subseteq \mathcal{U}_A$ and let $SN \in \mathcal{U}_{SN}$ be a system net. For any valid decomposition $D = \{SN^1, SN^2, ..., SN^n\} \in D(SN)$: $L$ is perfectly fitting system net $SN$ if and only if for all $1 \leq i \leq n : L \upharpoonright_{A_v(SN^i)}$ is perfectly fitting $SN^i$.*

The proof is available in [7]. By applying this theorem, we can compute the percentage of fitting traces in the overall event log. Let consider the fitting trace $\sigma_3 = \langle a, e, i, l, d, g, j, h, k, n, p, q\rangle$ and $SN_1$ (Figure 3.8). Under decomposition $D_1$, it is possible to obtain the subalignments by first projecting the trace on the subnets in Figure 3.9 and later aligning each subtrace with the corresponding subnet:

**Alignments 3.2.1** (Optimal alignment of $\sigma_3$)**.**

$$\gamma_3 = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|} a & \gg & e & i & l & d & g & j & h & k & n & p & q \\ \hline a & b & e & i & l & d & g & j & h & k & n & p & q \end{array}$$

**Alignments 3.2.2** (Optimal subalignments of $\sigma_3$)**.**

$$\gamma_3^1 = \begin{array}{|c|} a \\ \hline a \end{array} \quad \gamma_3^2 = \begin{array}{|c|c|} a & \gg \\ \hline a & b \end{array} \quad \gamma_3^3 = \begin{array}{|c|c|} a & d \\ \hline a & d \end{array} \quad \gamma_3^4 = \begin{array}{|c|c|c|c|} \gg & e & i & l \\ \hline b & e & i & l \end{array} \quad \gamma_3^5 = \begin{array}{|c|} \\ \hline \end{array}$$

$$\gamma_3^6 = \begin{array}{|c|c|c|c|c|c|} d & g & j & h & k & n \\ \hline d & g & j & h & k & n \end{array} \quad \gamma_3^7 = \begin{array}{|c|c|} l & p \\ \hline l & p \end{array} \quad \gamma_3^8 = \begin{array}{|c|c|} n & p \\ \hline n & p \end{array} \quad \gamma_3^9 = \begin{array}{|c|c|} p & q \\ \hline p & q \end{array}$$

Knowing just the percentage of traces fitting may be not sufficient. We should find a way to compute the cost of the overall alignments.

A naive approach to aggregate the results per subcomponent, would be to sum up all the misalignment costs of the subalignments under the standard cost function. For our example, if we sum the costs of the subalignments we would get a total of 2. However, the cost of the optimal alignment $\gamma_3$ is 1. The wrong result occurs because of border activity $b$ appears in two different subnets $SN_1^2$ and $SN_1^4$. For this reason, an adapted definition of cost function is required to avoid counting moves which involve border activities multiple times.

**Definition 3.2.3** (Adapted cost function)**.** Let $D = \{SN^1, SN^2, ..., SN^n\} \in D(SN)$ be a valid decomposition of some system net $SN$ and $\delta \in A_{LM} \to Q$ a cost function. The adapted cost function $\delta_D \in A_{LM} \to Q$ for decomposition D is defined as follows:

$$\delta_D(a,m) = \begin{cases} \frac{\delta(a,m)}{|SN_b(\alpha(a,m),D)|} & \text{if} \quad \alpha(a,m) \neq \tau \\ \delta(a,m) & \text{otherwise} \end{cases}$$

It follows the definition of an adapted fitness, hereafter referred as decomposed fitness metric.

**Definition 3.2.4** (Decomposed fitness metric)**.** Let $L \in B(A^*)$ be an event log and let $SN = (N, I, O) \in \mathcal{U}_{SN}$ be a system net with N $= (P, T, F, l)$. Let $D = \{SN^1, SN^2, ..., SN^n\} \in D(SN)$ be a valid decomposition of $SN$. For all $1 \leq i \leq n$, $SN^i = (N^i, I^i, O^i)$ is a subnet with an observable activity set $A_v^i = A_v(SN^i)$. For a log trace $\sigma_L \in L, \sigma_L^i = \sigma_L \upharpoonright_{A_v^i}$ is the projection of $\sigma_L$ on the activity set of subnet $SN^i$.

$$fit_D(\sigma_L, SN, \delta) = 1 - \frac{\sum_{i\in\{1,\ldots,n\}} \delta_D(\lambda(\sigma_L^i, SN^i))}{move_M(SN) + move_L(\sigma_L)}$$

For an event log L, its decomposed fitness metric is computed as follows:

$$fit_D(L, SN, \delta) = 1 - \frac{\sum_{\sigma_L\in L}\sum_{i\in\{1,\ldots,n\}} \delta_D(\lambda(\sigma_L^i, SN^i))}{|L| * move_M(SN) + \sum_{\sigma_L\in L} move_L(\sigma_L)}$$

In the decomposed fitness metric, the misalignment costs of each subalignments are first summed using the adapted cost function. Then, the total is normalized using the same value as the undecomposed relative fitness metric so that both metric values are normalized in the same manner. Let us consider again the example above of trace $\sigma_3$. The decomposed fitness metric is computed as $fit_D(\sigma_3, SN_1, \delta_1) = \frac{23}{24} \approx 0,958$, that is identical of the fitness metric of the overall trace and net.
However, this is not guaranteed with just this premises. In fact, local results do not consider the total structure of the net. Consequently, the minimum cost alignment for a subnet may appear particular expensive for the entire net. In paper [7] is proved that the decomposed fitness metric provides an upper bound to the fitness computing using the full alignment and the overall log and net. Let us see another example. We always consider the net in Figure 3.8 and the valid decomposition $D_1$ in Figure 3.9, but we use another trace $\sigma_4 = \langle a, b, e, i, l, d, g, h, n, j, k, p, q \rangle$. The optimal alignment and optimal subalignments are as Alignment 3.2.3 and 3.2.4:

**Alignments 3.2.3** (Optimal alignment of $\sigma_4$)**.**

$\gamma_4 =$

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $a$ | $b$ | $e$ | $i$ | $l$ | $d$ | $g$ | $h$ | $n$ | $j$ | $k$ | $\gg$ | $p$ | $q$ |
| $a$ | $b$ | $e$ | $i$ | $l$ | $d$ | $g$ | $h$ | $\gg$ | $j$ | $k$ | $n$ | $p$ | $q$ |

**Alignments 3.2.4** (Optimal subalignments).

$$\gamma_4^1 = \begin{array}{|c|} a \\ \hline a \end{array} \quad \gamma_4^2 = \begin{array}{|c|c|} a & b \\ \hline a & b \end{array} \quad \gamma_4^3 = \begin{array}{|c|c|} a & d \\ \hline a & d \end{array} \quad \gamma_4^4 = \begin{array}{|c|c|c|c|} b & e & i & l \\ \hline b & e & i & l \end{array} \quad \gamma_4^5 = \begin{array}{|c|} \\ \hline \end{array}$$

$$\gamma_4^6 = \begin{array}{|c|c|c|c|c|c|} d & g & h & n & k & \gg \\ \hline d & g & j & \gg & k & n \end{array} \quad \gamma_4^7 = \begin{array}{|c|c|} l & p \\ \hline l & p \end{array} \quad \gamma_4^8 = \begin{array}{|c|c|} n & p \\ \hline n & p \end{array} \quad \gamma_4^9 = \begin{array}{|c|c|} p & q \\ \hline p & q \end{array}$$

The cost of the overall alignment $\gamma_4$ is equal 2. Instead, applying the adapted cost function the total cost of the subalignments is 1, that is a lower bound according to costs and consequently an upper bound for fitness. This happens because the moves involving border activity $n$ are not identical between subalignments $\gamma_4^6$ and $\gamma_4^8$; the moves involving border activity $n$ in the two subalignments are not in agreement and we can see a *conflict*.

To overcome this problem and to compute exactly fitness, in [1] is defined the property of *border agreement*: sequences of moves involving the same border activity have to be in agreement across all subalignments.

**Definition 3.2.5** (Border agreement). Let $L \in B(A^*)$ be an event log and let $SN = (N, I, O) \in \mathcal{U}_{SN}$ be a system net with $N = (P, T, F, l)$. Let $D = SN^1, SN^2, ..., SN^n \in D(SN)$be a valid decomposition of $SN$. For all $1 \leq i \leq n$, $SN^i = (N^i, I^i, O^i)$ is a subnet with an observable activity set $A_v^i = A_v(SN^i)$.
For a border activity $a \in A_b(D)$, let $a_{LM} = \{(a, (a, t)), (\gg, (a, t)), (a, \gg)\}$ be the set of legal moves for activity $a$, where $t \in T$ such that $l(t) = a$. Let $SN^i \in SN_b(a, D)$ be a subnet that has the border activity $a$. For a log trace $\sigma_L \in L$, $\sigma_L^i = \sigma_L \upharpoonright_{A_v^i}$ is the projection of $\sigma_L$ on the activity set of $SN^i$. $\gamma_i \in A_{LM}^*$ denotes an optimal alignment between the sublog trace $\sigma_L i$ and some complete firing sequence of a subnet $\sigma_M^i \in \phi_f(SN^i)$.
The set of subalignments $\gamma^1, ..., \gamma^n$ are under border agreement on a border activity $a \in A_b(D)$ if, and only if, $\gamma^i \upharpoonright_{a_{LM}} = \gamma^j \upharpoonright_{a_{LM}}$, for all $SN^i, SN^j \in SN^b(a, D)$. The set of subalignments $\gamma^1, ..., \gamma^n$ are under total border agreement (t.b.a.) if, and only if, border agreement is achieved one by one on all the border activities in $\gamma^1, ..., \gamma^n$ following the order of their occurrences across $\gamma^1, ..., \gamma^n$, starting with the first occurring border activity in subnet $SN^i \in D$ where $I^i \geq I$.

Given the properties of a sequence, there is border agreement if the following three conditions are satisfied:

* $\gamma^i \upharpoonright_{a_{LM}}$ has an equal number of moves as $\gamma^j \upharpoonright_{a_{LM}}$.
* $\gamma^i \upharpoonright_{a_{LM}}$ has the same move types as $\gamma^j \upharpoonright_{a_{LM}}$, i.e. if $\gamma^i \upharpoonright_{a_{LM}}$ has one log move, then $\gamma^j \upharpoonright_{a_{LM}}$ must also have one log move.
* The order of moves in $\gamma^i \upharpoonright_{a_{LM}}$and $\gamma^j \upharpoonright_{a_{LM}}$ are the same.

Applying this definition, it is easy to see that subalignments of trace $\sigma_4$ are not under total border agreement.

If the total border agreement is not satisfied, the decomposed fitness metric coincides with an upper bound of the overall fitness. Instead, when the border agreement property is satisfied, the decomposed fitness metric corresponds exactly with the overall metric.

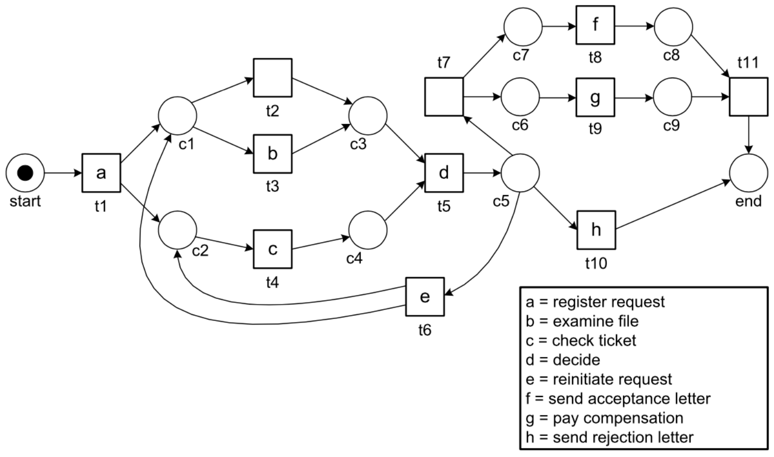


**Figure 3.10:** Labeled Petri net. Source: [7]

**Theorem 3.2.2** (Exact value for decomposed fitness metric under total border agreement)**.** *Let $L \in B(A^*)$ be an event log and let $\sigma_L \in L$ be a log trace. Let $SN = (N, I, O) \in \mathcal{U}_{SN}$ be a system net and let $D = \{SN^1, SN^2, ..., SN^n\} \in D(SN)$be a valid decomposition of SN. For all $1 \leq i \leq n$, $SN^i = (N^i, I^i, O^i)$ is a subnet with an observable activity set $A_v^i = A_v(SN^i)$.$\sigma_L^1, ..., \sigma_L^n$ are the subtraces from the projection of $\sigma_L$ onto the activity sets of $SN^1, ..., SN^n$ such that $\sigma_L^i = \sigma_L \upharpoonright_{A_v^i}$. $\gamma^i$ is an optimal subalignment between $\sigma_L^i$ and some complete firing sequence of the corresponding subnet $\sigma_M^i \in \phi_f(SN^i)$.*
*Let $\gamma^1, ..., \gamma^n$ be the set of subalignments and let them be under total border agreement. The decomposed fitness metric computed using this set of subalignments equals the relative fitness metric computed with the overall alignment between $\sigma_L$ and $SN$:*

$$fit(\sigma_L, SN, \delta) = fit_D(\sigma_L, SN, \delta)$$

*For the log L, if for all the log traces in L, their corresponding set of subalignments is under total border agreement,*

$$fit(L, SN, \delta) = fit_D(L, SN, \delta)$$

The proof of this theorem is available in [1].

### 3.2.2 Decomposition Algorithms for Splitting Petri Nets

After presenting the theory behind decomposing Petri nets, we present two of the most wide-spreadly employed algorithms to split models that guarantee the result to be a valid decomposition.

**Maximal Decomposition** The idea and the implementation of this algorithm are presented in [7]. The aim is to split the original net into individual subnets as small as possible. An example of a maximal decomposition of the model in Figure 3.10 is depicted in Figure 3.11. This approach assumes the original system net to be fully connected. This is not limiting: isolated transitions or places can be removed as they do not influence the behavior of the original net. By removing isolated nodes, the

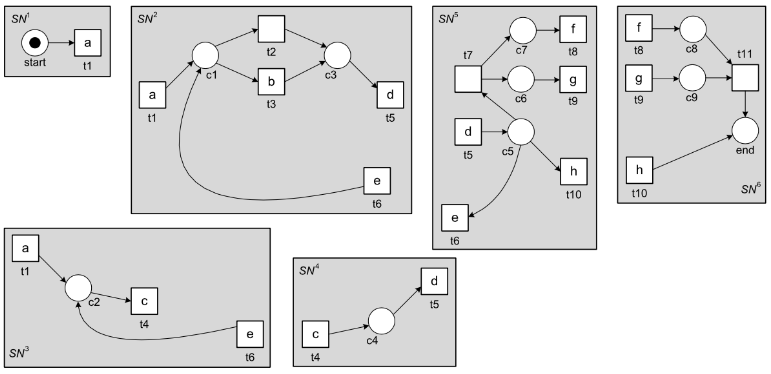


**Figure 3.11:** Maximal decomposition of the system net shown in 3.10. Source: [7]

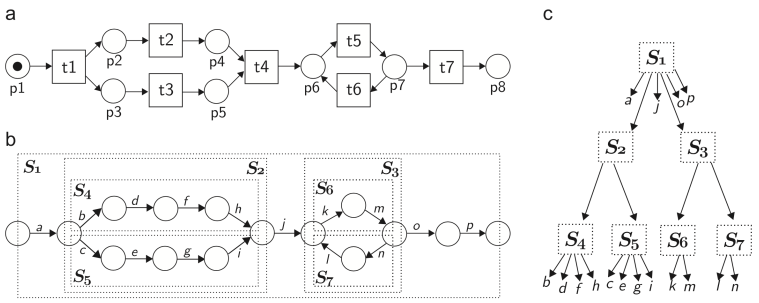


**Figure 3.12:** A Petri net, its workflow graph and the RPST and SESE decomposition. (a) Petri net. (b) Workflow Graph and SESE decomposition. (c) RPST. Source: [7]

construction of maximal decomposition is simplified.

The creation of the maximal decomposition is based on partitioning the edges which connect via undirected path involving as few as possible transitions/places, including invisible transition. After that, the sets that share non-unique observable activities are merged. Based on the new partitioning of the edges, subnets are created. In this way the requirement that visible transitions with the same label must reside in one subnet of valid decomposition is satisfied. Consider the Petri net in figure 3.10: $[[(t1, c2)]] = \{(t1, c2), (c2, t4), (t6, c2)\}$. $(c2, t4)$ and $(t6, c2)$ are added to $[[(t1, c2)]]$ because $c2$ is not a visible transition. $(t6, c1)$ and $(c5, t6)$ are not added to $[[(t1, c2)]]$ because $t6$ is a visible transition. In our example, all visible transitions have a unique label, so we can directly create subnets by the partitioning of the edges. From $[[(t1, c2)]]$ is created $SN^3$.

In [7] is proved that maximal decomposition is a valid decomposition, we leave this proof to the reader as in this thesis is used the next algorithm to split the model.

**Single-Entry Single-Exit (SESE) decomposition** Another important decomposition algorithm is based on the idea of *Single-Entry Single-Exit (SESE)*. In [6] the authors proposed the algorithm for constructing the *Refined Process Structure Tree*

*(RPST)*, i.e., a hierarchical structure containing all the canonical SESEs of a model. First, we define what is a SESE. SESE is a special type of subgraph with a very restricted interface with respect to the rest of the graph.

**Definition 3.2.6** (SESE)**.** A set of edges $S \subseteq E$ is a SESE (Single-Exit-Single-Entry) of graph $G = (V, E)$ iff $G_S$ has exactly two boundary nodes: one entry and one exit. A SESE is trivial if it is composed of a single edge. $S$ is a canonical SESE of $G$ if it does not partially overlap with any other SESE of $G$, i.e., given any other SESE $S'$ of G, they are nested ($S \subseteq S'$ or $S' \subseteq S$) or they are disjoint ($S \cup S' = \emptyset$).

The starting point of SESE decomposition is to consider Petri net as a *worflow graph*. A worflow graph is a directed graph where no distinctions are made between places and transitions. Afterwards, it is possible to compute the RPST, that is the tree composed of the set of all the canonical SESEs of a workflow net, such that, the parent of a canonical SESE $S$ is the smallest canonical SESE that contains $S$. The root of the tree is the entire graph, and the leaves are the trivial SESEs.
In [8], the computation of the RPST is significantly simplified and generalized reducing the implementation effort considerably. In Figure 3.12 it is possible to see an example of a Petri net, its corresponding workflow net and its RPST.

Intuitively, each SESE may represent a subprocess within the main process (i.e., the interior nodes are not connected with the rest of the net), and the analysis of every SESE can be performed independently. This means we can use the RPST of a net to select a possible set of SESEs, forming a decomposition. Carmona et al. introduce and define a SESE decomposition [9]:

**Definition 3.2.7** (SESE decomposition)**.** Consider the RPST decomposition of WF-net, where $\mathbb{S}$ represents all the SESEs in the RPST. We define a *transverse-cut* over the RPST as a set of SESEs $\mathbb{D} \subseteq \mathbb{S}$ such that any path from the root to a leaf of RPST contains one and only one SESE in $\mathbb{D}$. Given a transverse-cut $\mathbb{D} = \{S^1; S^2; ...S^n\}$, let the decomposition $\mathbb{D}_D$ be defined as $\mathbb{D}_D = \{SN^{S1}; SN^{S2}; ...SN^{Sn}\}$, where $SN^{Si}$ is the Petri net determined by the SESE $Si$, and the projection of the initial and final markings on the places of the subnet.

Algorithm 1 generates a decomposition which limits the maximum size of each component to a fixed threshold of activities $k$, details how to perform a SESE decompostion and in order to control the size and complexity of individual items, according to the use. The algorithm keeps a set of nodes that conforms the decomposition (D) and a set of nodes to consider (V). Initially V contains the roof of RPST, which is the entire net. Then, the algorithm checks, for each node $v$ to consider, if the number of arcs of SESE $v$ is less than or equal to k. If this is the case, $v$ is included in the decomposition, otherwise, $v$ is discarded and the RPST children of $v$ is included into the nodes to consider. The algorithm proposed has linear complexity with respect to the size of the RPST, and termination is guaranteed by the fact that the size of the component is reduced in every iteration.

A decomposition based directly on SESEs is not necessarily a valid decomposition. This is because we do not distinguish between places and transitions. It may happen that a place becomes a border node or that boundary places/transitions may be shared among subnets. To overcome this problem we introduce the concept of *bridging*. This technique consists of: transforming each place boundary found into a transition boundary, creating explicit subnets (called *bridges*) for each boundary place. The

**Algorithm 1** k-decomposition algorithm

```
procedure K-DEC(RPST, k)                ▷
    $V = \{root(RPST)\}$
    $D = 0$
    while $V \neq \emptyset$ do
        $V \leftarrow pop(V)$
        if $|v.arcs()| \leq k$ then
            $D = D \bigcup \{v\}$
        else
            $V = V \bigcup \{children(v)\}$
        end if
    end while
    return D
end procedure
```

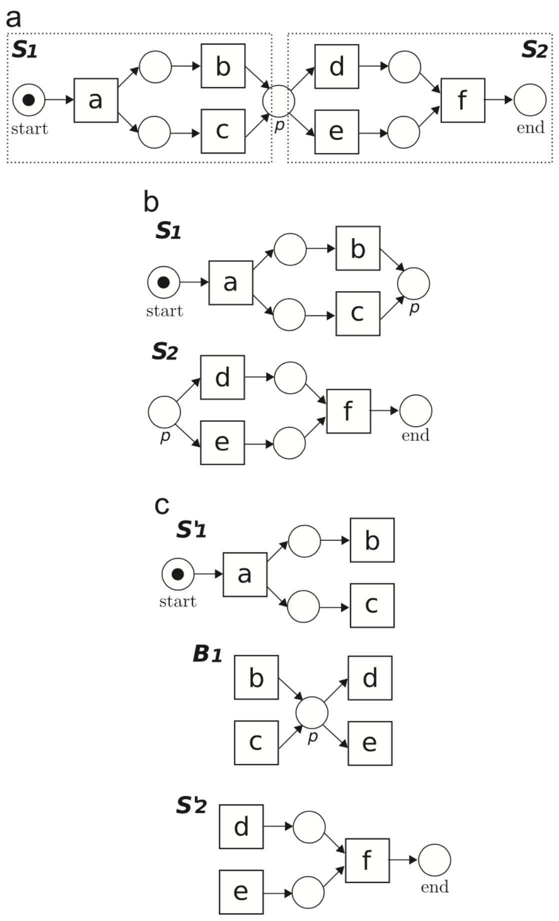


**Figure 3.13:** Example of bridging technique. (a) Petri net with a SESE decomposition. (b) A SESE decomposition. (c) A SESE decomposition with bridging. Source: [9]

bridges contain all the transitions connected with the boundary place, and they are in charge of keeping the place synchronized among subnets. In addition, the boundary places together with the arcs connected to them are removed from the original subnets. We present an example in Figure 3.13. In this case, the boundary place $p$ is removed from $S_1$ and $S_2$. Then the bridge $B_1$ is created. The resulting decomposition is a valid decomposition. In [9] it is proved that SESE decomposition using bridging is a valid decomposition.
In this thesis, we use this algorithm to perform decomposition. We discuss how it has been integrated in our algorithm in Chapter 4.

### 3.2.3 Decomposition-Based Algorithms for Conformance Checking

In previous section we define how to split Petri nets in order to compute alignments in each of the sub model, and how to do this in a valid way. Now we present complete *divide and conquer* approaches that first split the net and event log into fragments, then they compute alignments, and finally they merge the results to compute the overall conformance.

#### Pseudo Alignments

Sometimes it is sufficient an approximation of the fitness between alignments and process model. For this reason the plug-in *"Decomposed Replay Algorithm"* is being introduced. The plug-in has been implemented in the DecomposedReplayer package of ProM 6. The algorithm takes as inputs the model depicted as an Accepting Petri net[2] and the event log.

First, the model and log are separated according to one of decomposition algorithms chose by the user, producing an array of sub-models and one of sub logs. For each of them, the alignments are computed. Fractions of the model are treated as independent Petri net, so for each subpetri is called the method that compute the replay. At this moment, the result is an array of subalignments needed to be merged together. In [10] is presented the procedure used to merge subalignments into optimal alignments if it is possible, otherwise it produces so-called *pseudo alignments.* We describe briefly this algorithm as it is also used in our approach.

First, let introduce the concept of *pseudo alignment.* For arbitrary decomposed alignments, a merge trace alignment may not exist. This happens if the property of total border agreement is not satisfied. In other words this occurs if in subalignments are present some conflicts. In such cases, the result of merging those subalignments is a so-called pseudo alignment. Using these pseudo-alignments, the algorithm can handle conflicts between the decomposed alignments. A pseudo alignment is constructed in a pessimistic way:

∗ in case of activity conflicts, the most expensive of the conflicting legal moves is added to the resulting pseudo alignment;

∗ in case of model move conflicts, one of these model moves is selected, and added to the pseudo-alignment.

[2]An Accepting Petri net is a Petri which is specified also the set of final markings instead of just the initial marking.

In [10] it is proved that a pseudo alignment always exists, and it may not coincide with a valid alignment. However, having as result a pseudo alignment may be useful as its cost represent a lower bound for the overall alignment, consequently its fitness is an upper bound. The algorithm proceeds by applying three alignment stitching rules, followed by two pseudo-alignment stitching rules. The alignment stitching rules construct a merged trace alignment, if possible. If this succeeds, we know that the result is again an alignment, and that the reported costs are exact, and not just a lower bound. Otherwise, we need a pseudo-alignment stitching rule to be able to continue.

**Definition 3.2.8** (Stitching function Y ). Let $\mathcal{H}$ be the set of all possible trace pseudo-alignments of $L$ and $N$, and let $\mathcal{H}^i$ be the set of all possible trace alignments of $L^i$ and $N^i$. The function $Y \in (\mathcal{H} \times A^* \times \mathcal{H}^1 \times ... \times \mathcal{H}^n) \rightarrow \mathcal{H}$ returns the first argument concatenated by the merged trace pseudo-alignment of the third and following arguments $(h^1, ..., h^n)$, where the second argument $(\sigma)$ is used to guide the stitching. As a result, $Y(\langle\rangle, \sigma, h^1, ..., h^n)$ returns the merged trace pseudo-alignment of $(h^1, ..., h^n)$.

Let describe the first alignment stitching rules:

* **Alignment Stitching Rule 1 (All done):** it is a simple rule that detects when the algorithm is done.
* **Alignment Stitching Rule 2 (Activity w/o conflict):** this rule allows the algorithm to continue if all relevant decomposed alignments agree on the first activity in the trace. If so, this activity is now dealt with and so are the corresponding legal moves in the relevant decomposed alignments. For the irrelevant decomposed alignments, nothing changes.
* **Alignment Stitching Rule 3 (Transition w/o conflict):** in this case the algorithm continues if all relevant decomposed alignments agree on a next model move. If so, these legal moves are now dealt with.

As already written, applying these rules will result in an alignment if the algorithm ends and Rule 1 can be applied. However, it may be that no rule is applicable before reaching the end of one or more decomposed alignments.
If some conflicts are present the following two rules are applied. However, the result will be a pseudo alignment.

* **Pseudo-alignment Stitching Rule 1 (Activity w/ conflict):** this rule allows the algorithm to continue if the relevant decomposed alignments disagree on the next legal move containing the first activity in the trace, that is, a synchronous or log move. If so, the most expensive of the conflicting legal moves is added to the resulting pseudo alignment, the activity in the trace is now dealt with, and so are all the conflicting moves in the relevant decomposed alignment.
* **Pseudo-alignment Stitching Rule 2 (Transition w/ conflict):** this rule that allows the algorithm to continue if the relevant decomposed alignments disagree on a next model move. If so, one of these model moves is selected, and added to the pseudo-alignment, and all corresponding model moves are now dealt with.

This procedure has been implemented in the LogAlignment package of ProM 6, and it is used in *"Decomposed Replay Algorithm"* plug-in.

The Decomposed Replay algorithm returns the results of this merging procedure. Consequently, the resulting fitness is an upper bound of the overall alignments.

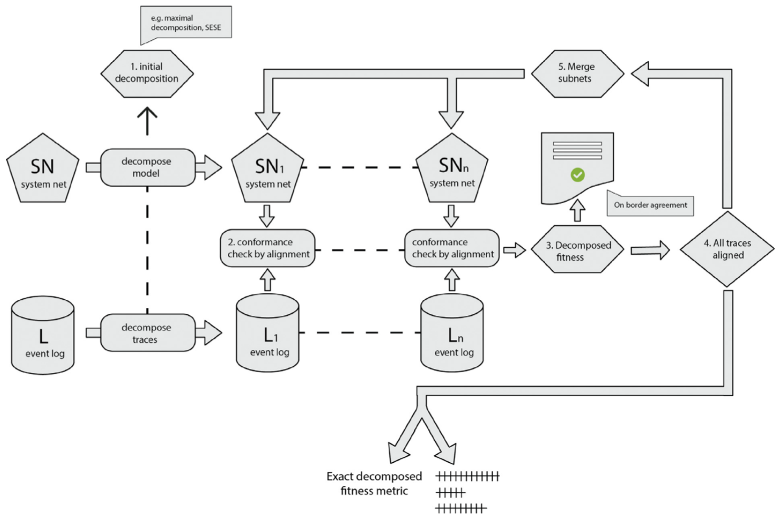


**Figure 3.14:** Overview of the Recomposing Replay Algorithm. Source: [1]

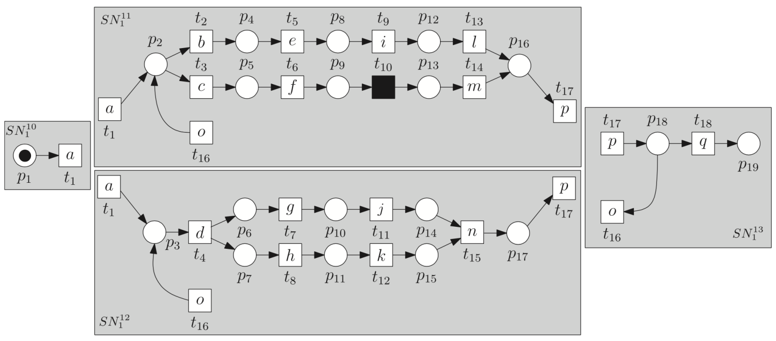


**Figure 3.15:** New decomposition $D_2$ after one recomposing interation. Source: [1]

## Recomposing Replay Algorithm

As written above, the algorithm to compute pseudo-alignments produces just an approximation of the overall fitness of a process model. In [1], authors present a novel method to compute the exact overall result by using the property of total border agreement (Definition 3.2.5): *Recomposing Conformance.*

The main idea is to iterative merge the subnets with conflicts in just one subnet and perform again the Conformance Checking. In this way the disagreeing subnets can not have conflicts anymore.

As illustrated in Figure 3.14, the first step is to decompose the Petri Net using a valid decomposition. This can be performed using the splitting algorithms we have seen before *Maximal Decomposition* or *Single-Entry Single-Exit Approach.* Following this initial decomposition, the log is projected into the subnets obtaining the corresponding sublogs. At this point, subalignments are computed.

In step 3 of fig. 3.14, the decomposed fitness metric is calculated. The fitness values for those alignments computed under total agreement are recorded. Consequently, the associated traces are marked as done, as it has been possible to compute the exact fitness value. For the remaining traces, we need to resolve the border disagreements using the recomposing approach.

Let consider the Petri net in Figure 3.8, its initial decomposition $D_1$ in fig. 3.9 and the trace $\sigma_5 = \langle a, b, e, i, d, g, j, h, k, b, e, i, l, d, g, j, h, k, n, p, q \rangle$. Alignment 3.2.5 depicts the resulting subalignments:

**Alignments 3.2.5.**

$$\gamma_5^1 = \begin{array}{|c|} a \\ \hline a \end{array} \quad \gamma_5^2 = \begin{array}{|c|c|c|} a & b & b \\ \hline a & b & \gg \end{array} \quad \gamma_5^3 = \begin{array}{|c|c|c|} a & d & d \\ \hline a & d & \gg \end{array} \quad \gamma_5^4 = \begin{array}{|c|c|c|c|c|c|c|c|} b & e & i & \gg & b & e & i & l \\ \hline b & e & i & l & b & e & i & l \end{array}$$

$$\gamma_5^5 = \begin{array}{|c|} \\ \hline \\ \end{array} \quad \gamma_5^6 = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|} d & g & j & h & k & \gg & d & g & j & h & k & n \\ \hline d & g & j & h & k & n & d & g & j & h & k & n \end{array}$$

$$\gamma_5^7 = \begin{array}{|c|c|} l & p \\ \hline l & p \end{array} \quad \gamma_5^8 = \begin{array}{|c|c|} n & p \\ \hline n & p \end{array} \quad \gamma_5^9 = \begin{array}{|c|c|} p & q \\ \hline p & q \end{array}$$

In this subalignments we can detect border agreement problems for activities $b, d, l$, and $n$. The algorithm can select one of more activities that cause these problems, and merge the corresponding subnets. In our example, the subnets $SN_1^2, SN_1^3, SN_1^4, SN_1^5$, $SN_1^6, SN_1^7$ and $SN_1^8$ are recomposed so that activity b, d, l, and n are no longer border activities. This produces a new decomposition $D_2$ figured in 3.15. At this point, a new Conformance Checking is performed using the new decomposition and the corresponding sublogs. The new subalignments are the following:

**Alignments 3.2.6.**

$$\gamma_5^{10} = \begin{array}{|c|} a \\ \hline a \end{array} \quad \gamma_5^{12} = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|} a & d & g & j & h & k & \gg & \gg & \gg & d & g & j & h & k & n & p \\ \hline a & d & g & j & h & k & n & p & o & d & g & j & h & k & n & p \end{array}$$

$$\gamma_5^{11} = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|} a & b & e & i & \gg & \gg & \gg & b & e & i & l & p \\ \hline a & b & e & i & l & p & o & b & e & i & l & p \end{array} \quad \gamma_5^{13} = \begin{array}{|c|c|} p & q \\ \hline p & q \end{array}$$

However, the results can not be merged together again as total border agreement is not satisfied. We need to perform recomposition another time. In this case, the problematic activity is $p$. Similarly to previous iteration, the algorithm recomposes the subnets which present $p$ and produce a new decomposition. After this, the total border agreement property is satisfied, so it is possible to compute the decomposed fitness value that will correspond to the exact fitness value.
The algorithm can be implemented with some termination conditions:

* all log traces have been either aligned under total border agreement or have been rejected because of the number of border conflicts is over a given threshold;

* the overall time threshold has been surpassed;

∗ having aligned a target percentage of traces in the log under total border agreement or the overall fitness interval value is narrow enough;

∗ the maximum number of iterations is reached.

In this case, the result will be a fitness interval. These settings can be modified while starting the plug-in.

This algorithm has been implemented as the plug-in *Recomposing Replay Algorithm* in the DecomposedReplayer package of ProM6.7. As this approach is the newest decomposing technique, in Chapter 5 we compare our algorithm with this one.

# Chapter 4

# Greedy Approach to Compute Alignments of Process Models and Events Logs

*In this chapter, our new approach to perform Conformance Checking using decomposition is described. A deeper explanation of the main idea and implementation is provide.*

In previous chapter, we have described two algorithms for performing Conformance Checking: *Pseudo Alignments* and *Recomposing Replay Algorithm*. Both of these algorithms offer advantages in the problem of computing alignments. In [10] and [1] some experiments are presented which show these approaches improve computation time respect to the classic algorithm existing at that moment. However, both approaches have certain limitations. The pseudo-alignments approach produces just an estimation of the overall fitness, which may not always be accurate. On the other hand, the Recomposing Replay Algorithm yields true alignments as the final result. Nevertheless, for complex model, it may require several iterations to resolve border disagreements between subnets. This could lead to longer computation times due to the enormous number of recomposing steps needed to be pass through. Additionally, recomposed subnets may be as large as the original model. Furthermore, when processing a trace in the log, if the number of border conflicts exceeds a prefixed threshold, all sub-alignments are immediately merged, resulting in a pseudo-alignment.
To address these issues, we tried to contribute by creating a new decomposition-based algorithm. In this chapter, we first describe how this approach works and the underlying idea. Subsequently, we present the implementation and discuss the problems encountered during the deployment.

## 4.1 Algorithm

We have learned that achieving total border agreement is necessary to compute exact fitness value. This implies that subalignments must not have conflicts in their moves during calculation. Our objective is to develop an algorithm that does not require multiple recompositions of the subnets to resolve conflicts, in order of reducing computation time.

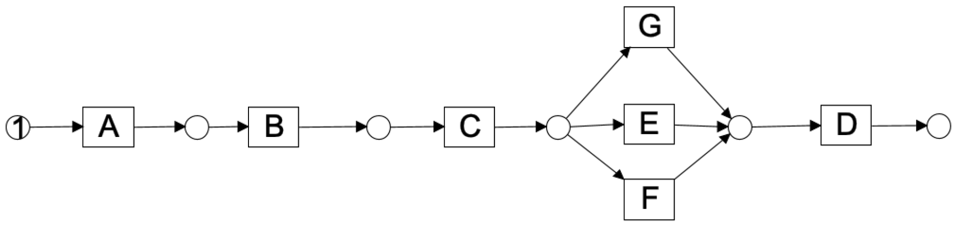


**Figure 4.1:** System Net $SN_2$

The idea is to maintain the initial decomposition of the Petri net and iteratively change subalignments to satisfy the total border agreement condition. In other words, for each trace, we compute the optimal subalignment of the corresponding subnet. If conflicts arise, one of the subalignments is chosen, and its *next-best subalignment* is computed. This process is repeated until no more conflicts are present or a predefined threshold number of iterations is reached. In the latter case, total border agreement is not satisfied, resulting in a pseudo alignment.
Our motivation for using *next-best subalignments* stems from the computational time expense associated with the replay method. Therefore, our approach aims to save time during this phase in two ways:

∗ the replay method involves instantiating other objects as configuration before performing replay. While this configuration differs for each subnet, it remains the same for every trace. We store these configurations before calculating optimal subalignments, eliminating the need to compute them for each trace in the log.

∗ In *Recomposing Replay Algorithm* replay is done at every iteration as the decomposition of the net changes. In our approach, we compute optimal subalignments only once for each trace and store them. The next-best subalignment operation since the replay has already been calculated. This approach is expected to speed up the process, as the iterations would be very rapid.

First, we denote what is the **next-best subalignment**. In Chapter 3, we provided the definition of optimal alignments and highlighted that they are not unique. For the same trace, multiple alignments with different costs exist. Let imagine ordering the multitude of possible alignments from the optimal ones to the worst. Once we have computed the optimal one, we can descend through this structure until we reach the worst alignment. Therefore, given a subalignment, the *next-best subalignment* is the next subalignment in our ordered list, its cost is equal or greater than the original one.

Now, we present the pseudo-code of our algorithm, and then we explain the various phases in detail.

The algorithm is essentially structured in three phases: splitting net and log, computing subalignments and finding/resolving conflicts, merging subalignments.

**Algorithm 2** Greedy Approach for Decomposed Replay

```
    Input: System Net $SN$, event log $L$
    Output: The set $A$ = set of alignments between the event log $L$ and the System Net $SN$
  procedure Greedy Approach for Decomposed Replay Algorithm
      $A$ :=empty
      Split SN into a valid decomposition using SESE approach, i.e. $D = \{SN^1, ..., SN^n\} \in D(SN)$
      Split L according to D, i.e. $DL = \{DL^1, ..., DL^n\}$
      for each trace $\sigma_k \in L$ do
          Compute $\Gamma_k := (\gamma_k^i)_{i \in \{1,...,n\}}$ optimal subalignments between subnets D and the projected subtraces $DL_k = \{DL_k^1, ..., DL_k^n\}$
          while conflicts are present do
              $\gamma_k^{MAX}$ := subalignment with the major number of conflicts
              $\gamma_k^{MAX'}$ := NEXTALIGNMENT($\gamma_k^{MAX}$)
              Update $\Gamma_k$ with the new $\gamma_k^{MAX'}$
          end while
          Merge subalignments into new alignment $\gamma_k$
          ADD(A, $\gamma_k$)
      end for
      return A
  end procedure
```

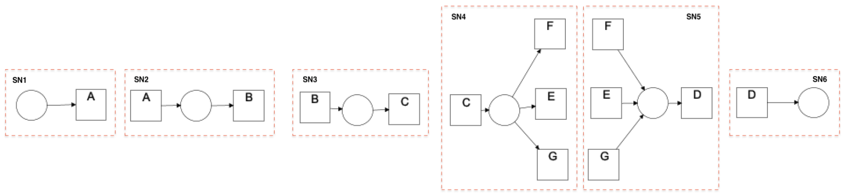


**Figure 4.2:** Decomposition $D_2$ of the Petri net $SN_2$ depicted in Figure 4.1

### Phase1 : Splitting Petri Net and Event Log

As outlined in Algorithm 2, the initial steps involve decomposing the net and the log. The decomposition of the system net must adhere to the properties of a valid decomposition; otherwise, Conformance Checking would not be feasible (see Section 3.2). We use the *SESE-based decomposition* to split the Petri net.

Following the initial decomposition, the log is projected onto the subnets of the decomposition to obtain sublogs. At this point, we have an array of subnets and an array of lists of subtraces. Implementation details will be discussed in the next section.

Consider the Petri Net $SN_2$ depicted in Figure 4.1. After the splitting procedure, the result is the decomposition $D_2$ depicted in Figure 4.2. For the sake of simplicity, we consider the event log $L_2$ with just one trace $\sigma_0 = \langle A, G, D\rangle$. The decomposed event log is $DL_2 = \{\sigma_0^1 = \langle A\rangle, \sigma_0^2 = \langle A\rangle, \sigma_0^3 = \langle\rangle, \sigma_0^4 = \langle G\rangle, \sigma_0^5 = \langle G, D\rangle, \sigma_0^6 = \langle D\rangle\}$, that is the projection of the trace onto the decomposition.

**Phase2 : Computing Subalignments and Resolving Conflicts**

At this point, for each trace of the log we compute the optimal subalignments. If they are under total border agreement condition, the algorithm skips to next phase, and it merges the result. Otherwise, we need to find which subalignment has the greatest number of conflicts. Let compute the optimal subalignments of the trace $\sigma_0$ of our example:

**Alignments 4.1.1** (Subalignments of trace $\sigma_0$: iteration 0)**.**

$$\gamma_0^1 = \begin{array}{|c|} A \\ \hline A \end{array} \quad \gamma_0^2 = \begin{array}{|c|c|} A & \gg \\ \hline A & B \end{array} \quad \gamma_0^3 = \begin{array}{|c|} \gg \\ \hline \gg \end{array}$$

$$\gamma_0^4 = \begin{array}{|c|} G \\ \hline \gg \end{array} \quad \gamma_0^5 = \begin{array}{|c|c|} G & D \\ \hline G & D \end{array} \quad \gamma_0^6 = \begin{array}{|c|} D \\ \hline D \end{array}$$

We can calculate the number of conflicts between other subalignments:

$$\gamma_0^1 = 0 \quad \gamma_0^2 = 1 \quad \gamma_0^3 = 1 \quad \gamma_0^4 = 1 \quad \gamma_0^5 = 1 \quad \gamma_0^6 = 0$$

The subalignments $\gamma_0^2$ and $\gamma_0^3$ are in conflict as the first one present the model move $B$, instead $\gamma_0^3$ is an empty alignment. Note that $\gamma_0^3$ is the alignment of subnet $SN3$ which presents the border activity $B$. This means that it should not be empty.
The other conflict is caused by activity $G$, as $\gamma_0^5$ presents a synchronous move, instead $\gamma_0^4$ a log move.
It is not unusual that there are more alignments with the same number of conflicts. The heuristic to select which one to compute the next-best subalignment is the following:

* the function returns the subalignments with the maximum number of conflicts if it is unique;
* otherwise from the set of subalignments that has the max number of conflicts it returns:
  - the alignment with just log moves;
  - or the alignment with the minimum cost.

The use of a heuristic is naturally greedy and may not return optimal alignments. In our example, subalignment $\gamma_0^4$ is selected, so its next-best subalignment is computed. The new combination becomes the following:

**Alignments 4.1.2** (Subalignments of trace $\sigma_0$: iteration 1)**.**

$$\gamma_0^1 = \begin{array}{|c|} A \\ \hline A \end{array} \quad \gamma_0^2 = \begin{array}{|c|c|} A & \gg \\ \hline A & B \end{array} \quad \gamma_0^3 = \begin{array}{|c|} \gg \\ \hline \gg \end{array}$$

$$\gamma_0^4 = \begin{array}{|c|c|} \gg & G \\ \hline C & G \end{array} \quad \gamma_0^5 = \begin{array}{|c|c|} G & D \\ \hline G & D \end{array} \quad \gamma_0^6 = \begin{array}{|c|} D \\ \hline D \end{array}$$

It is clear that conflict caused by activity $G$ has been solved, as we have changed the alignment $\gamma_0^4$. Although, conflict between activity $B$ is still present and one with conflict $C$ appears. Thus, we need to perform another iteration. This time the number of conflicts between other subalignments are:

$$\gamma_0^1 = 0 \quad \gamma_0^2 = 1 \quad \gamma_0^3 = 2 \quad \gamma_0^4 = 1 \quad \gamma_0^5 = 0 \quad \gamma_0^6 = 0$$

Intuitively, we compute the next-best subalignment of $\gamma_0^3$ as it owns the greatest number of conflicts. The new combination is:

**Alignments 4.1.3** (Subalignments of trace $\sigma_0$: iteration 2)**.**

$$\gamma_0^1 = \begin{array}{|c|} A \\ \hline A \end{array} \quad \gamma_0^2 = \begin{array}{|c|c|} A & \gg \\ \hline A & B \end{array} \quad \gamma_0^3 = \begin{array}{|c|c|} \gg & \gg \\ \hline B & C \end{array}$$

$$\gamma_0^4 = \begin{array}{|c|c|} \gg & G \\ \hline C & G \end{array} \quad \gamma_0^5 = \begin{array}{|c|c|} G & D \\ \hline G & D \end{array} \quad \gamma_0^6 = \begin{array}{|c|} D \\ \hline D \end{array}$$

At this point, the algorithm checks another time if some conflicts are present. The conflict with activity $B$ is being solved, and no new conflicts have arisen. This means the subalignments satisfy total border agreement condition and we can go to the next phase.

### Phase3: Merging subalignments

At this stage, we are able to merge subalignments into a new alignment. This is achieved by the stitching rules proposed in *Pseudo Alignments'* algorithm in Section 3.2.3. Resuming, if no more conflicts are present in subalignments, i.e. total border agreement condition is satisfied, we can apply the first three *Alignment Stitching Rules*, and producing an alignment. After applying the merge algorithm, the overall alignment of our example is:

$$\gamma_0 = \begin{array}{|c|c|c|c|c|} A & \gg & \gg & G & D \\ \hline A & B & C & G & D \end{array}$$

Finally, we can calculate the cost of the resulting alignment. Note that it is not used the adapted cost function for decomposed approach as we use the algorithm to merge subalignments into an alignment. The result is stored in set $A$ which contains the overall alignment for the event log $L$. The algorithm starts again to **Phase2** with the next trace of the event log. When alignments of all traces have been computed, they are returned to the user, providing the overall fitness value and the number of iterations each trace performs.

The result of our algorithm is shown in Figures 4.3 and 4.4. Figure 4.3 shows how the moves of all alignments are projected onto the Petri net model. This allows us to quickly gain an insight into frequencies of occurrences of activities and deviations. Each transition is filled in with a blue color whose intensity grows with the number of moves for that transition. Inside each transition, below the activity name two numbers are shown in brackets (x/y) which indicate that the alignments contain x synchronous moves and y model moves for the transition. Figures 4.4 enumerates the set of resulting

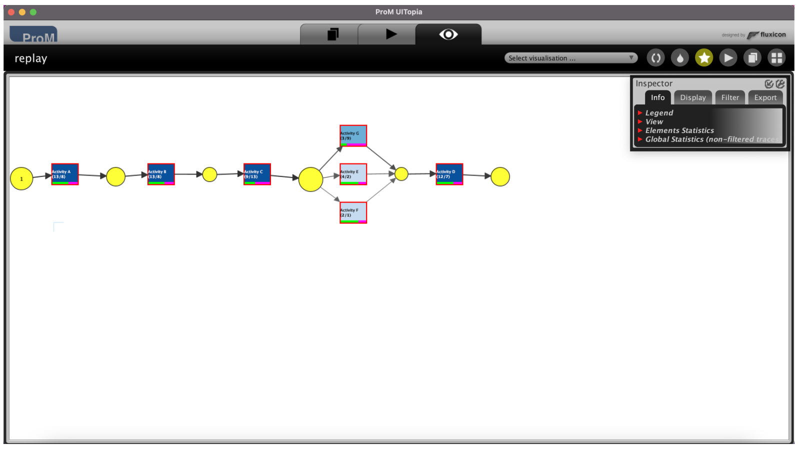
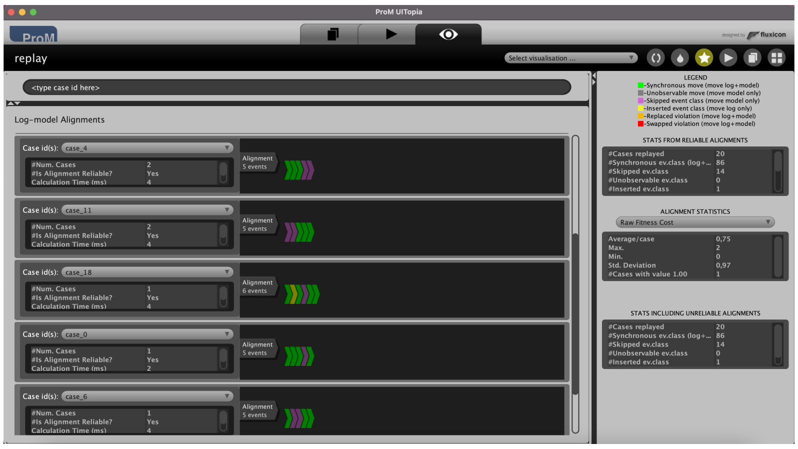


**Figure 4.3:** Visualization of the result of our approach implemented in ProM. Here the moves of all alignments are projected into occurrences of activities.



| Case id(s) | | | Alignment |
|---|---|---|---|
| case_4 | #Num. Cases | 2 | 5 events |
| | #Is Alignment Reliable? | Yes | |
| | Calculation Time (ms) | 4 | |
| case_11 | #Num. Cases | 2 | 5 events |
| | #Is Alignment Reliable? | Yes | |
| | Calculation Time (ms) | 4 | |
| case_18 | #Num. Cases | 1 | 6 events |
| | #Is Alignment Reliable? | Yes | |
| | Calculation Time (ms) | 4 | |
| case_0 | #Num. Cases | 1 | 5 events |
| | #Is Alignment Reliable? | Yes | |
| | Calculation Time (ms) | 2 | |
| case_6 | #Num. Cases | 1 | 5 events |
| | #Is Alignment Reliable? | Yes | |
| | Calculation Time (ms) | 4 | |

LEGEND
- 🟩-Synchronous move (move log+model)
- ⬜-Unobservable move (move model only)
- 🟪-Skipped event class (move model only)
- 🟨-Inserted event class (move log only)
- 🟧-Replaced violation (move log+model)
- 🟥-Swapped violation (move log+model)

STATS FROM RELIABLE ALIGNMENTS

| | |
|---|---|
| #Cases replayed | 20 |
| #Synchronous ev.class (log+... | 86 |
| #Skipped ev.class | 14 |
| #Unobservable ev.class | 0 |
| #Inserted ev.class | 1 |

ALIGNMENT STATISTICS

Raw Fitness Cost

| | |
|---|---|
| Average/case | 0,75 |
| Max. | 2 |
| Min. | 0 |
| Std. Deviation | 0,97 |
| #Cases with value 1.00 | 1 |

STATS INCLUDING UNRELIABLE ALIGNMENTS

| | |
|---|---|
| #Cases replayed | 20 |
| #Synchronous ev.class (log+... | 86 |
| #Skipped ev.class | 14 |
| #Unobservable ev.class | 0 |
| #Inserted ev.class | 1 |

**Figure 4.4:** Other visualization of the result of our approach implemented in ProM. The plug-in provides the alignments computed using our approach. The green moves are synchronous moves, the purple ones are model moves instead yellow ones are log moves.

alignments computed with our approach, with the relative cost and information.

Note that our approach is guarantee to only return optimal alignments for perfect fitting traces. Instead, if traces are not perfectly fitting we do not guarantee for optimal alignments, as computing just the next-best alignment we can choose a combination without conflict, but with a cost that is greater than the optimal one (recall the greedy nature of the approach).

### 4.1.1 Improved Algorithm

The termination of our algorithm is provided when no more conflicts are present among the subalignments. Consequently, it may end after an enormous number of iterations. Moreover, just computing the next-best alignment it is difficult to find the right sequence of subalignments without testing all the possible combinations.

In fact, in some cases the algorithm chooses the wrong subalignment to start the searching. That happens because of the heuristic component of this technique.
We use as parameter of selection the number of conflicts. However, this does not ensure that the chosen subalignment is the one that need to change in order to solve border disagreement.

Considering this problem, we improve the algorithm with the purpose of having more accurate results without spending too much time in iterations.
In the previous example, we assume to save the set of subalignments as a list, where every time the next-best subalignment is computed, the list is updated by replacing the old alignment with the new one. In this way, we do not take track of each alignment computed. Instead of a simple list, we use an array of lists. The length of the array is the number of subnets. Each list contains the subalignments calculated so far, such that the one in the first position is the optimal. When the next-best subalignment is calculated, it is added to the corresponding list. Thus, in the last position of each list, the newest subalignments are present, and they are used to performing **Phase2**.
Consider the Petri net in Figure 4.1, its decomposition in Figure 4.2 and the new trace $\sigma_1 = \langle B, A, G, D \rangle$. After **Phase1** we obtain the following optimal subalignments:

**Alignments 4.1.4** (Subalignments of trace $\sigma_1$: iteration 0)**.**

$$\gamma_1^1 = \begin{array}{|c|} A \\ \hline A \end{array} \quad \gamma_1^2 = \begin{array}{|c|c|c|} B & A & \gg \\ \hline \gg & A & B \end{array} \quad \gamma_1^3 = \begin{array}{|c|c|} B & \gg \\ \hline B & C \end{array}$$

$$\gamma_1^4 = \begin{array}{|c|c|} \gg & G \\ \hline C & G \end{array} \quad \gamma_1^5 = \begin{array}{|c|c|} G & D \\ \hline G & D \end{array} \quad \gamma_1^6 = \begin{array}{|c|} D \\ \hline D \end{array}$$

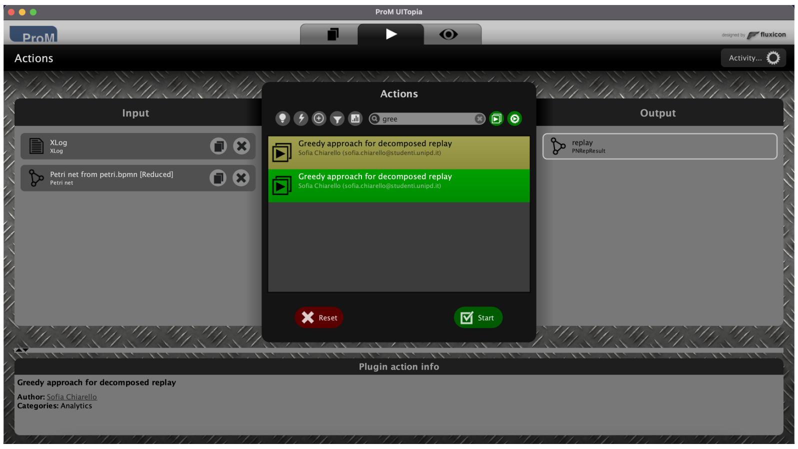
The number of conflicts for each alignment are:

$$\gamma_1^1 = 0 \quad \gamma_1^2 = 2 \quad \gamma_1^3 = 2 \quad \gamma_1^4 = 0 \quad \gamma_1^5 = 0 \quad \gamma_1^6 = 0$$

The first subalignment selected is $\gamma_1^3$ as it owns the max number of conflicts and its cost is smaller respect to $\gamma_1^2$.
The algorithm continues iterating for $x$ times. If after that the right combination has not been found yet, we restart the searching. The first explored subalignment, in this case $\gamma_1^3$, is blocked and it can not be selected. This because it is the right item to achieve a combination without disagreements.

At this point, the algorithm uses the same heuristic to select another alignment. Its next-optimal alignment may already have been calculated and saved in the data structure. In this manner, it is not necessary to calculate the next-best alignment again, as we have already done this operation in the previous phase. Then it checks the conflicts and iteration continues.
Back to our example, $\gamma_1^2$ is selected and the next-best optimal alignment is:

$$\gamma_1^2 = \begin{array}{|c|c|c|} \gg & B & A \\ \hline A & B & \gg \end{array}$$


ProM UITopia
ProM
designed by fluxicon
Actions
Activity...
Input
XLog
XLog
Petri net from petri.bpmn [Reduced]
Petri net
Actions
gree
Greedy approach for decomposed replay
Sofia Chiarello (sofia.chiarello@studenti.unipd.it)
Greedy approach for decomposed replay
Sofia Chiarello (sofia.chiarello@studenti.unipd.it)
Reset
Start
Output
replay
PNRepResult
Plugin action info

**Greedy approach for decomposed replay**

**Author:** Sofia Chiarello
**Categories:** Analytics

**Figure 4.5:** Visualization of ProM to select our algorithm

This solves the border disagreements as conflicts caused by activity $B$ are no more present, so the algorithm delivers this sequence to the merging method.
To improve the algorithm and to extend the number of combinations to check, it is possible to change the blocked subalignments in two different ways:

* if after $j$ iterations conflicts have not yet been resolved, the algorithm blocks the next-best subalignments of the first blocked. This is because the right subalignment may be skipped during the searching phase.
* If after freezing $y$ subalignments of the same subnets we have not yet reached the total border agreement condition, we change the blocked alignment, with someone with another index.

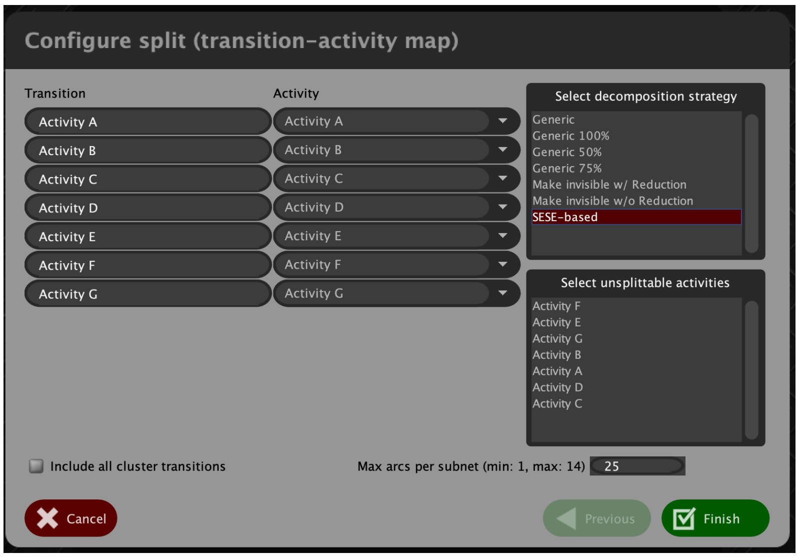
In this way it is possible to explore more combination in the searching space, without exponentially increasing time since most of the alignments has been already calculated. If after a prefixed threshold of iterations we have not been able to find a correct combination, the result is a pseudo alignment. In chapter 5 we discuss the results of our approach, even in terms of number of alignments and subalignments computed.

## 4.2 Implementation

This algorithm has been implemented as a plug-in in the Process Mining software *ProM*. In Figure 4.5 we can see the visualization to start the plug-in. It takes in input a Petri net model and the related event log. Then it produces the replay as output, in other words the total alignments computed using our method (an example of the visualization is in Figure 4.4).
In the previous section we present the idea and the procedure behind our approach. Now we explain in detail the implementation and how we assume to save computational time respect to existing algorithms.

### 4.2.1 Splitting Petri Net and Event Log

The first step of the algorithm is to split the two inputs. In the first place, we handle the Petri net.


Configure split (transition–activity map)

| Transition | Activity |
|---|---|
| Activity A | Activity A |
| Activity B | Activity B |
| Activity C | Activity C |
| Activity D | Activity D |
| Activity E | Activity E |
| Activity F | Activity F |
| Activity G | Activity G |

Select decomposition strategy
- Generic
- Generic 100%
- Generic 50%
- Generic 75%
- Make invisible w/ Reduction
- Make invisible w/o Reduction
- [✓ SESE-based]

Select unsplittable activities
- Activity F
- Activity E
- Activity G
- Activity B
- Activity A
- Activity D
- Activity C

☐ Include all cluster transitions

Max arcs per subnet (min: 1, max: 14) 25

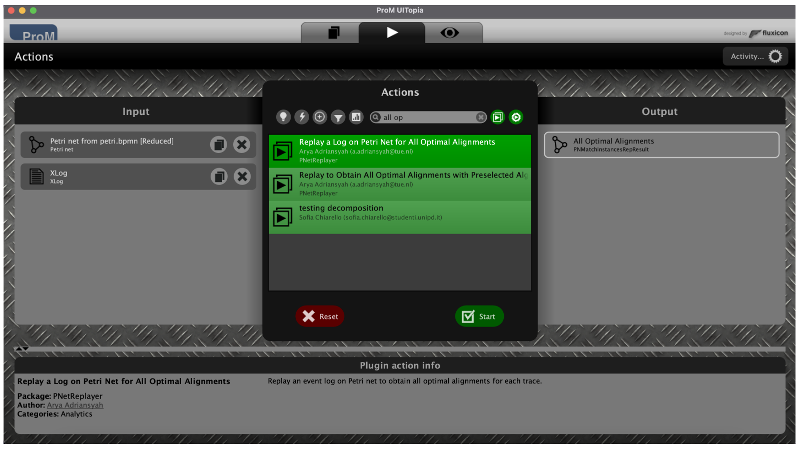
**Figure 4.6:** Visualization of the configuration for splitting a Petri net from the plug-in *"Split Accepting Petri Net"*. On the right is possible to select which decomposition strategy use.

*ProM* already provides some algorithms to deal this problem. In particular, we use the plug-in *"Split Accepting Petri Net"* which produces an array of subnets. The plug-in offers to configure the decomposition as it is depicted in Figure 4.6. It is possible to change the strategy to decompose the model and which activities to not split. In our algorithm, this plug-in is automatically called so the user does not have to select any configuration. The model is separated using SESE-based strategy and every activity can be decomposed by default. The return value is an array which each item is a subnet of the original Accepting Petri net.

At this point, we can split the event log using another existing plug-in: *"Split Event Log"*, which is also integrated in our code. It creates an array of sublogs. The first log in the array will contain the given log projected onto the activities of the first subnet in the model array, and so on.
In summary, after these operations we obtain two arrays: one containing the decomposition of the Petri net and the other sublogs for each subnets, i.e. each item $i$ of the array is a list of subtraces projected to the subnet $i$.

### 4.2.2 Computing Subalignments

The second phase of our approach is to compute subalignments for each subnets and then checking if some conflicts are present.
To perform replay and compute an alignment, *Prom* offers different strategies, one of them has been presented in the previous Chapter in 3.1.3. However, we need a procedure that can provide all the multitude of existing alignments, from the optimal to the worst, in an iterative way (we want to compute first the optimal one, and then when we asked that it returns the next-best). *"Replay a Log on Petri Net for All Optimal Alignments"* (Figure 4.7) accepts in input a Petri net model and the event log, it returns all the optimal alignments computed for each trace. Its configuration can be



Actions

Actions

all op

Petri net from petri.bpmn [Reduced]
Petri net

XLog
XLog

Replay a Log on Petri Net for All Optimal Alignments
Arya Adriansyah (a.adriansyah@tue.nl)
PNetReplayer

Replay to Obtain All Optimal Alignments with Preselected Alg
Arya Adriansyah (a.adriansyah@tue.nl)
PNetReplayer

testing decomposition
Sofia Chiarello (sofia.chiarello@studenti.unipd.it)

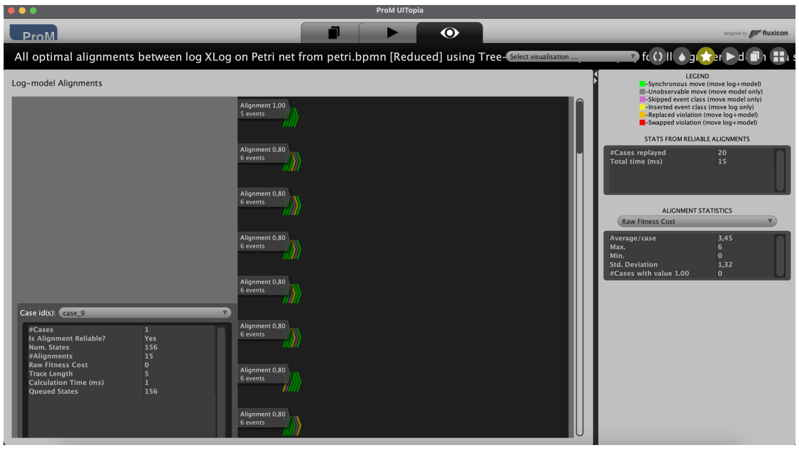
All Optimal Alignments
PNMatchInstancesRepResult

Plugin action info

**Replay a Log on Petri Net for All Optimal Alignments** Replay an event log on Petri net to obtain all optimal alignments for each trace.

**Package:** PNetReplayer
**Author:** Arya Adriansyah
**Categories:** Analytics

**Figure 4.7:** Visualization of the starting of the plug-in *"Replay a Log on Petri Net for All Optimal Alignments"* in Prom



All optimal alignments between log XLog on Petri net from petri.bpmn [Reduced] using Tree-

Log-model Alignments

Alignment 1,00
5 events

Alignment 0,80
6 events

Alignment 0,80
6 events

Alignment 0,80
6 events

Alignment 0,80
6 events

Alignment 0,80
6 events

Alignment 0,80
6 events

Alignment 0,80
6 events

Case id(s): case_9

| | |
|---|---|
| #Cases | 1 |
| Is Alignment Reliable? | Yes |
| Num. States | 156 |
| #Alignments | 15 |
| Raw Fitness Cost | 0 |
| Trace Length | 5 |
| Calculation Time (ms) | 1 |
| Queued States | 156 |

LEGEND
🟩-Synchronous move (move log+model)
⬛-Unobservable move (move model only)
🟪-Skipped event class (move model only)
🟨-Inserted event class (move log only)
🟧-Replaced violation (move log+model)
🟥-Swapped violation (move log+model)

STATS FROM RELIABLE ALIGNMENTS

| | |
|---|---|
| #Cases replayed | 20 |
| Total time (ms) | 15 |

ALIGNMENT STATISTICS

Raw Fitness Cost

| | |
|---|---|
| Average/case | 3,45 |
| Max. | 6 |
| Min. | 0 |
| Std. Deviation | 1,32 |
| #Cases with value 1.00 | 0 |

**Figure 4.8:** Visualization of the result of *"Replay a Log on Petri Net for All Optimal Alignments"* plug-in. It is computed all the possible alignments of trace *case_ 9* from fitness 1 to fitness equal to 0.8

changed in order to compute alignments down to a specific fitness value. In Figure 4.8 it is depicted the multitude of alignments for trace *case_ 9* from the best one (fitness value equal 1) to the alignment which its fitness coincide with the given threshold.

Intuitively the working of this plug-in is very similar to what we are looking for, thus we implemented a new algorithm that compute alignments in this way, but according to our needs. The new class is called NextAlignmentTreeAlg and essentially it calculates the replay. In order to compute the replay, this class needs a particular configuration that is different for each subnet, but it is the same for traces in the same sublog. This means that we can compute configurations just one time for each subnet and then use them when performing replay for each trace of the log.

### Creating Configurations to Perform Replay

After **Phase1** of our algorithm, for each subnet an instantiation of class NextAlignmentTreeAlg is created and the following parameters are setted:

* initial marking of the corresponding subnet;
* final markings of the corresponding subnet;
* function cost that maps each transition of the model;
* function cost that maps each event of the sublog;
* info and classifier of the event log.

In this way we store in memory every class ready to perform the replay when called with a subtrace. Then we can go to the next step without loosing time.

### Computing Subalignments

At this point, the environment for computing alignments has been setted and we can start to perform the replay. This operation is done for each trace of the log.
For each subnet, the corresponding instantiation of NextAlignmentTreeAlg calls its method replayLog(). It, essentially, takes the subnet, the subtrace and the already existing configuration, to compute the replay. Recalling from chapter 3, the calculation of a replay can be seen as a search problem. We use the **A* searching algorithm** to provide the cheapest firing sequence of the model. To compute the replay, it is used the AllOptAlignmentsTreeThread<PHead, DijkstraTail> which performs the searching and returns one (or more) alignments between a trace and a model. It provides the method getOptimalRecord(). On its first call, the A* algorithm is used to compute an optimal alignment between a model and a trace. Then it returns a Record, an object which stores little of information needed per volatile state in the A* Algorithm. Using this object, we can compute the resulting alignment.

The method getOptimalRecord() can be repeatedly calling to compute more alignments starting from the optimal one. This way, the algorithm guarantees that all states which are on a shortest path from the source to the target node (that in our case is the worst alignment) will be visited (note that this does not imply that all paths will be found). Back to our class, we instantiate a thread and then we call the first time the method getOptimalRecord(). The resulting Record is used to compute the alignment. The thread used is also stored in memory, as it will be used to compute the *next-best alignment*. It will be sufficient to call the getOptimalRecord() to calculate the next.

### Data Structure to Store Subalignments

At this moment, we have computed an optimal subalignment for each subnet and the corresponding thread. Optimal subalignments are stored in a data structure already mentioned in the previous Section (4.1.1). It is an array which length is the number of subnets. Each item of the array is a list which contains the subalignments computed so far. Consider the previous example in Figure 4.2 and trace $\sigma_0 = \langle A, G, D \rangle$. The optimal subalignments are stored in data structure as presented in Figure 4.9. Each

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| **0** | $\begin{array}{\|c\|} A \\ \hline A \end{array}$ | $\begin{array}{\|c\|c\|} A & \gg \\ \hline A & B \end{array}$ | $\begin{array}{\|c\|} \gg \\ \hline \gg \end{array}$ | $\begin{array}{\|c\|} G \\ \hline \gg \end{array}$ | $\begin{array}{\|c\|c\|} G & D \\ \hline G & D \end{array}$ | $\begin{array}{\|c\|} D \\ \hline D \end{array}$ |

**Figure 4.9:** Array of subalignments of trace $\gamma_0$ and Petri net $SN_2$. Each column corresponds to a subnet. In the first row are present the optimal subalignments computed.

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| **0** | $\begin{array}{\|c\|} A \\ \hline A \end{array}$ | $\begin{array}{\|c\|c\|} A & \gg \\ \hline A & B \end{array}$ | $\begin{array}{\|c\|} \gg \\ \hline \gg \end{array}$ | $\begin{array}{\|c\|} G \\ \hline \gg \end{array}$ | $\begin{array}{\|c\|c\|} G & D \\ \hline G & D \end{array}$ | $\begin{array}{\|c\|} D \\ \hline D \end{array}$ |
| **1** | | | | $\begin{array}{\|c\|c\|} \gg & G \\ \hline C & G \end{array}$ | | |

**Figure 4.10:** Array of subalignments of trace $\gamma_0$ and Petri net $SN_2$ after one iteration, i.e. computing the next-best subalignment operation. The subalignments of subnet number four is selected, so its next-best is stored in the second row.

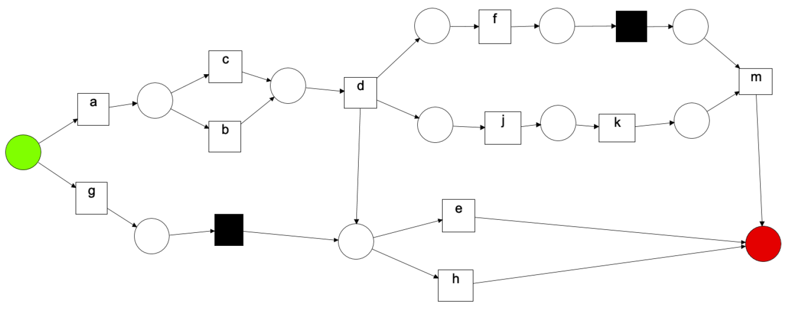


**Figure 4.11:** Petri net $SN_3$

column corresponds to a subnet. In the first row the optimal subalignments computed in the previous phase are present. After we checked conflicts and we selected the one alignment to perform the next, the data structure is updated by adding to the right list the next-best subalignment. In our example, subalignment $\gamma_0^4$ is selected, its next-best is added in the list as figured in 4.10. In the next iteration, the algorithm to check conflicts will use the configuration of item in position $[0;1]$, $[0;2]$, $[0;3]$, $[1;4]$, $[0;5]$, $[0;6]$, in other words the last item of each list.
Using this method of storing, we have all the history of computed subalignments and we can check more configurations as described in the improved version of the algorithm (Section 4.1.1).

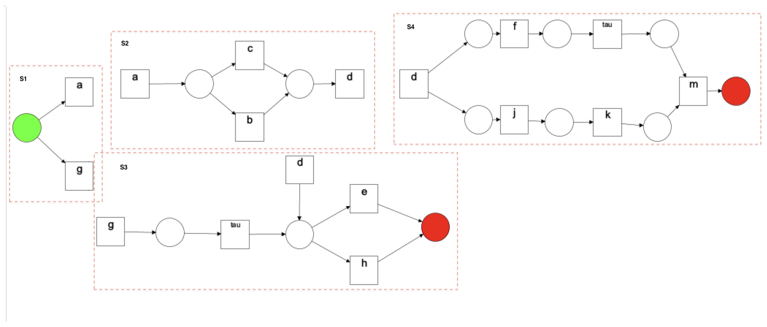


**Figure 4.12:** Decomposition $D_3$ of System Net $SN_3$ in picture 4.11.

### 4.2.3 Checking Conflicts

The next step is to iterative check if some conflicts are present between the subalignments and then select the one with the bigger number of disagreements. To achieve this, we implemented a method which taking the combination of subalignments, provides the number of conflicts for each subalignment.

For each subnet, the algorithm checks only the subalignments corresponding to the subnets which has some common activities with it. If we consider our example $SN_2$ and its decomposition in Figure 4.2, the subnet $SN1$ has the activity $A$ in common with just subnet $SN2$.
The number of conflicts for each subnet is the number of moves that are in disagreement. A move can be in disagreement just if it is a border activity. So for each subalignment we check just the activities that are in common with the other subnets.
Let call **current subalignment** the one alignment for which we are counting conflicts and **next subalignment** the subalignment that it is necessary to compare with, i.e. calculated from a subnet with border common transitions.

The first conflict we take in account is if the current subalignment is empty, but the next subalignment presents some moves corresponding to their common activities. This is the case of the previous example of trace $\sigma_0$ and its subalignments computed in 4.1.1. Let consider $\gamma_0^3$ as the current subalignment and as the next subalignment $\gamma_0^2$. $\gamma_0^3$ is an empty alignment, but $\gamma_0^2$ presents a model move for activity $B$ which is a common activity of their subnets $SN3$ and $SN2$. Consequently, the counter of conflicts is incremented as the number of disagreements.

Otherwise, if current subalignment is not empty, we can start to check its moves. To better understanding the mechanism, we use as example the more complex Petri net $SN_3$ in Figure 4.11 and its decomposition pictured in 4.12. Let compute the optimal subalignments of trace $\sigma_2 = \langle a, g, b, d, c, d, f, j, e, h, m \rangle$ :

**Alignments 4.2.1** (Subalignments of trace $\sigma_2$: iteration 0)**.**

$\gamma_2^1 =$

| | |
|---|---|
| $a$ | $g$ |
| $a$ | $\gg$ |

$\gamma_2^2 =$

| | | | | |
|---|---|---|---|---|
| a | b | d | c | d |
| a | b | d | c | $\gg$ |

$\gamma_2^3 =$

| | | | | |
|---|---|---|---|---|
| $g$ | d | d | e | $h$ |
| $\gg$ | d | d | e | $h$ |

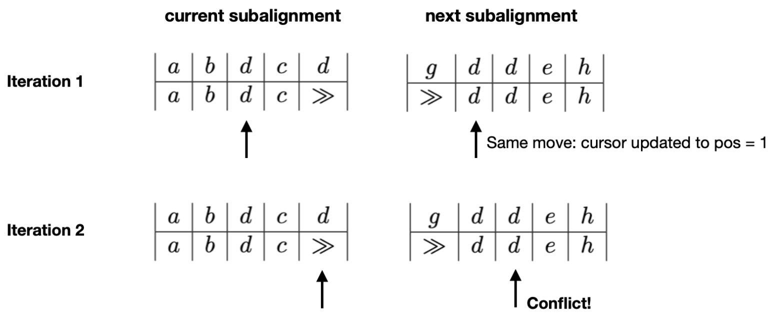


**Figure 4.13:** Representation of algorithm to check conflicts. In the first iteration, it checks the *d synchronous move*. As it is present also in the next subalignment, the cursor is updated to this position. In the following iteration, the *d log move* is checked. There is a conflict as the next alignment presents a *d synchronous move*.

$$\gamma_2^4 = \begin{array}{|c|c|c|c|c|c|c|} d & d & f & \tau & j & \gg & m \\ \hline d & \gg & f & \tau & j & \gg & m \end{array}$$

We try to compute the number of conflicts between $\gamma_2^2$ (current subalignment) and $\gamma_2^3$ (next subalignment). In Figure 4.13 we can see the steps of our algorithm. We start checking each move of the current subalignment that are in common with the next. In our case, the two subnets $S2$ and $S3$ share just activity $d$. The first activities of $\gamma_2^2$ are $a$ and $b$ that are not common activities, so we skip to check the *synchronous move* $d$. Now the algorithm searches in next subalignment a move with the same activity, if it is not of the same type, there is a conflict and the counter is incremented. In $\gamma_2^3$ the first move with a common activity is the *synchronous move d*. Its type is the same as the current move, so there is no conflict between these two moves. At this point we update the cursor of the next subalignment in order that it will point to the next move, and we can continue the search for the rest of the alignment as the correct move has been found. We pass to check the next move in $\gamma_2^2$ that is a *log move d*. The cursor of next subalignment points to the next common move that is a *synchronous move*. This is a conflict as it should have the same type of current move, so the counter is updated. Note that in this way, we consider conflict also if common activities are in the wrong position. If in $\gamma_2^3$ were a log $d$ move and then a synchronous move, the algorithm would count two conflicts.

However this method is not sufficient to find every possible conflicts. We also check if some common activities are present in the current subalignment but not in the next and vice versa.

In conclusion, the method returns the numbers of conflicts calculated for each subalignments. They are used from the heuristic function to select for which subalignment compute the next-best. At each iteration, this method is called. If some conflicts are present, we select one subalignment and compute its next-best just calling the function GETOPTIMALRECORD() described above. The subalignments data structure is updated, so the conflicts are checked again using the new combination.

### 4.2.4 Merging Subalignments

As already mention, the final step is to merge the sequence of subalignments into just one alignment. This is done after achieving the right combination of subalignments without border disagreements, or when the number of iterations reached a specific threshold.

To merge subalignments, the plug-in *"Merge Log Alignments"* is used which implements the five *Stitching Rules* described in Section 3.2.3. This plug-in as been adapted to our needs and it is embedded in our code. The main modification is that it works for trace and not for the event log. Thus, it produces one alignment from the set of subalignments, and it is called for each trace of the event log.

The resulted alignment may be a pseudo alignment if the algorithm stops before to solve all conflicts.

## 4.3 Implementation Problems Encountered

During the implementation of the algorithm we encountered different problems we tried to solve. In this section we enumerate these issues and their related solutions.

### 4.3.1 Empty Subtraces

The decomposition of an event log may produce empty subtraces according to the trace. Back to our example using Petri net $SN_2$ in Figure 4.1 and the trace $\sigma_0 = \langle A, G, D \rangle$, the decomposed trace is the set of subtraces $DL_2 = \{\sigma_0^1 = \langle A \rangle, \sigma_0^2 = \langle A \rangle, \sigma_0^3 = \langle \rangle, \sigma_0^4 = \langle G \rangle, \sigma_0^5 = \langle G, D \rangle, \sigma_0^6 = \langle D \rangle\}$. Subtrace $\sigma_0^3$ is empty as the overall trace does not present any of the activities of the corresponding subnets $SN3$.

The plug-in *"Replay a Log on Petri Net for All Optimal Alignments"* on which we rely, always only return an empty alignment (i.e. an alignment with no moves) when the original trace is empty (i.e. containing no events). However, there are some cases in which this is not correct. We recall the combination for the trace $\sigma_0$ after one iteration (this is the same of 4.1.2):

$$\gamma_0^1 = \begin{array}{|c|} A \\ \hline A \end{array} \quad \gamma_0^2 = \begin{array}{|c|c|} A & \gg \\ \hline A & B \end{array} \quad \gamma_0^3 = \begin{array}{|c|} \gg \\ \hline \gg \end{array}$$

$$\gamma_0^4 = \begin{array}{|c|c|} \gg & G \\ \hline C & G \end{array} \quad \gamma_0^5 = \begin{array}{|c|c|} G & D \\ \hline G & D \end{array} \quad \gamma_0^6 = \begin{array}{|c|} D \\ \hline D \end{array}$$

As already explained, the conflict here is caused by activities $B$ and $C$ in subalignments $\gamma_0^2$, $\gamma_0^3$ and $\gamma_0^4$. Intuitively, $\gamma_0^3$ should present model moves of transition $B$ and $C$ to compute the correct alignment and to solve all conflicts. However, as the subtrace is an empty trace, the replay will never compute a no-empty alignment, so we need to fix this bug.

Our idea is to modify the structure of the subnets, adding a fictions transition. In this way, when there are no problems in computing alignments, this transition

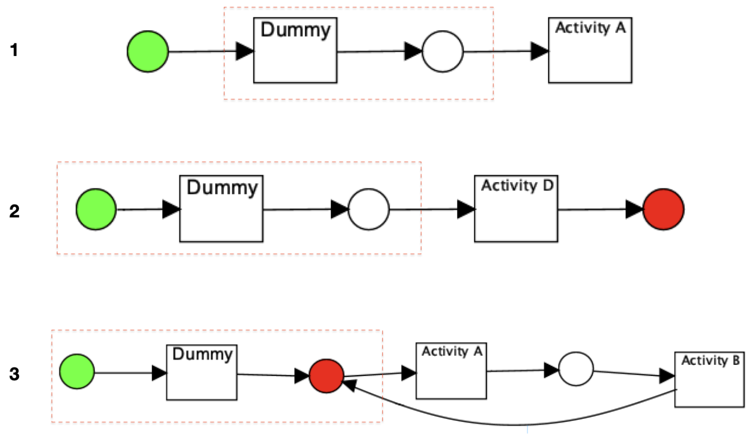


**Figure 4.14:** Example of three subnets of the System net $SN_2$ modified in order to solve the issue of empty alignments. In (1) it is pictured the initial subnet; in (2) the final subnet and in (3) the generic case. The added part is highlighted.

move is removed from the subalignment. If an empty alignment is selected to calculate its next-best (i.e. a non-empty alignment is expected to be returned), the subtrace is modified adding the fictions transition. In this way, the replay algorithm is forced to produce an alignment. In conclusion, the fictions transition move is removed.

Our idea is to modify the structure of the subnets, adding a fictions transition "Dummy". Every time we produce an alignment, the move corresponding to this fictions transition is removed from the alignment. If an empty alignment is selected to calculate its next-best (i.e. a non-empty alignment is expected to be returned), the subtrace is modified by adding the event "Dummy" which is mapped to the fictions transition. In this way, the replay is forced to produce a non-empty alignment as the trace is not empty anymore. Note that an empty subalignment is not always wrong. It may happen that it does not cause any conflict.

The subnets are modified in this way according to their nature (we refer to Figure 4.14 for explanation):

∗ if the subnet is the initial part of the original net, we add just the transition *"Dummy"* after the initial place, and it is connected to the first activity (subnet **1**);

∗ if the subnet has a final place, the highlighted part in subnet **2** is added to the original one;

∗ otherwise, the generic subnet is modified by adding the highlighted part in subnet **3**. The red place is marked as final, and it is reachable from the last transitions. In this way it is possible to achieve an empty alignment and to loop to the beginning of the original subnet if it is necessary.

### 4.3.2 Memory Issue

In ProM, there are ad-hoc data structures designed to store the resulting alignments. However, these objects are large in terms of memory occupation. Since our algorithm needs to compute and retain in memory a large number of alignments, the available memory can quickly be filled up, leading to *out-of-memory* errors.

To address this issue, the ProM heavy alignment object representation is replaced by a matrix of STRINGS with two rows and N columns, where N is the number of moves in the alignment to be represented. The first row corresponds to the log moves, and the second row represents the model moves. In the case of synchronous moves, the name of the activity, represented as a String object, will be present in both rows. The symbol $\gg$ is encoded as a NULL STRING pointer.

# Chapter 5

# Experiments

*In this chapter are present the experiments to test the goodness of our algorithm respect to the existing ones. First we describe the preliminary steps to produce our dataset, then we provide the results of our experiments. In conclusion we discuss the results and the future works regarding our approach*

This chapter is dedicated to experiments aimed at verifying the effectiveness of our new algorithm described in the previous chapter. Specifically, our goal is to test the algorithm by providing it with input from large process models, as the decomposed Conformance Checking methods are particular beneficial in these cases. Furthermore, we aim to compare its results with existing Conformance Checking plug-ins, utilizing *"Replay a log on Petri Net for conformance analysis"* as a classic approach and *"Recomposing Replay Algorithm"* for comparison with a decomposition-based technique.
Before presenting the results, we will describe the Process Mining tools used in this thesis and outline the preliminary steps taken to create the dataset. Finally, we will provide a discussion of the experiments.

## 5.1 Preliminary Steps

The experiments were conducted using a dataset consisting of different process models to represent the variability of possible scenarios. The dataset includes synthetically generated process models and their corresponding event logs. Ten processes were randomly generated using the PLG2 tool, encompassing large processes containing combinations of common workflow patterns such as XOR, AND, loops, and invisible transitions. *Process Log Generator (PLG2)* is an open-source software that can generate random process models based on general complexity parameters, such as the maximum depth, the maximum number of AND or OR branches a model can have. The model is generated as a BPMN model but can be easily converted into a Petri net. PLG2 is also capable executing a given process model to generate event logs. Event logs can be produced with any number of traces. Additionally, the software provides the option to adjust the value of noise in logs. Our dataset for experiments was generated using this tool. However, to introduce the noise factor, another technique was utilized. We will describe in detail how we generate our process models and event logs.

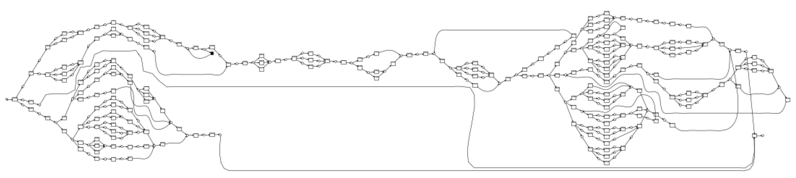


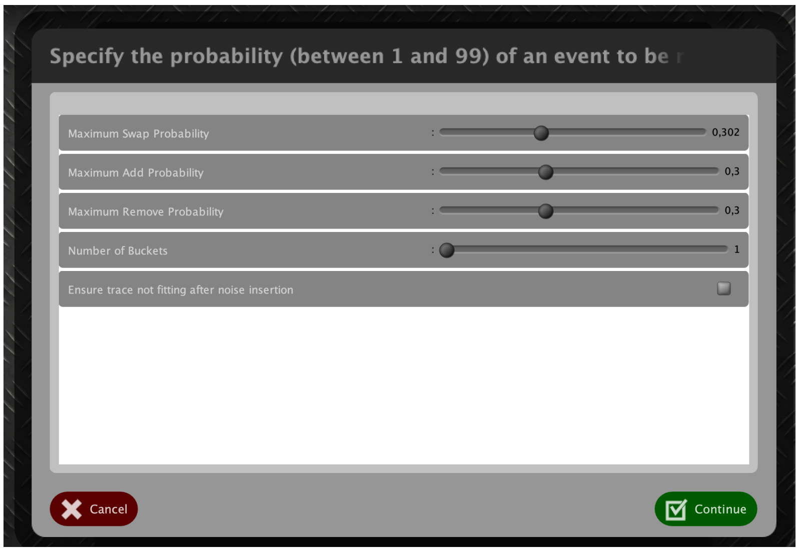
**Figure 5.1:** Petri Net P0 of dataset. It presents 206 activities and 92 gateways.

**Table 5.1:** Characteristics of the synthetic nets.

| Name | # Activities | # Getaways | Initial Decomposition |
|---|---|---|---|
| P0 | 43 | 22 | 20 |
| P1 | 67 | 32 | 22 |
| P2 | 101 | 50 | 30 |
| P3 | 106 | 50 | 42 |
| P4 | 116 | 50 | 41 |
| P5 | 128 | 56 | 47 |
| P6 | 133 | 32 | 43 |
| P7 | 140 | 54 | 58 |
| P8 | 191 | 76 | 87 |
| P9 | 206 | 92 | 75 |

The characteristics of the models are shown in Table 5.1. The size of the models generated were progressively increased to challenge the algorithms under evaluation, with the largest model consisting of 206 activities and 92 gateways, as depicted in Figure 5.1. Logs were generated from the models using simulation, and various operations were applied to emulate different plausible noise scenarios. Each log has **1000 cases**.

Initially, no-noise logs were generated using the PLG2 tool. *No-noise* logs are perfectly fitting with the corresponding models, indicating that all the behavior observed in the log can be matched to the behavior modeled by the model. As a result, the log can be replayed perfectly on the model, and the fitness value equals to 1. Note that this is not particularly significant for our results, as the subalignments computed would always be under the total border agreement condition, and no iterations would be required. Logs with noise were produced using the ProM plug-in *"Add, Swap and Remove Events"*. This plug-in takes a Petri net model and the corresponding event log as input. Noise can be introduced at different levels by **adding**, **swapping** and **removing** events from the trace. The plug-in allows modification of the maximum probability of introducing noise, ranging from 0 to 0.8. In Figure 5.2 the noise configuration is illustrated. Our logs were modified using all three parameters to achieve noises with a maximum probability of **6%** and **14%**.



Specify the probability (between 1 and 99) of an event to be

| | |
|---|---|
| Maximum Swap Probability : | 0,302 |
| Maximum Add Probability : | 0,3 |
| Maximum Remove Probability : | 0,3 |
| Number of Buckets : | 1 |
| Ensure trace not fitting after noise insertion | ☐ |

**Figure 5.2:** Visualization of the noise configuration of plug-in *"Add, Swap and Remove Events"* implemented in ProM.

## 5.2 Results

Our algorithm has been tested on a desktop with an Intel Core i7-10875H processor, 32 GB RAM, running Windows 10 Pro and using a 64-bit version of Java 8.
In this section, we present the results of our experiments, starting with the simplest scenario where the model and log are perfectly fitting, and then introducing noise factors. We tested the improved version of our algorithms (Section 4.1.1), which may return pseudo-alignments. The fitness values computed are only for those traces for which an alignment was computed. We will compare our results with the implementation for the approach that does not employ decomposition, and for the approach based on Recomposition (see Section 3.1.3 and 3.2.3).

### 5.2.1 Noiseless Scenario

We initially conducted experiments with the simplest scenario where the model and log are perfectly fitting. In this section, we present Conformance Checking results for logs without noise ("no-noise" logs). The objectives of this section are:

* to compare and analyze computation times of replays under the three different approaches;
* to compare the fitness values of our greedy approach with respect to the other two methods.

The results of our algorithm are shown in Table 5.2. Here, we observe that the time used to compute alignments increases with the size of models, as expected, because it must compute more subalignments. Since there is no noise in our event logs, all the fitness values are equal to 1. In *Conflicts* column, the numbers of traces with conflicts are shown, i.e. the algorithm goes through iterations computing the next-best subalignment to resolve border disagreements. As expected, no trace presents conflicts,

**Table 5.2:** Results about Greedy Approach for Decomposed Replay in noiseless scenario.

| Greedy Approach | | | | | |
|---|---|---|---|---|---|
| **Name** | **Activities** | **Time (s)** | **Fitness** | **Conflicts** | **# Pseudo Alignments** |
| P0 | 43 | 5,1 | 1 | 0 | 0 |
| P1 | 67 | 6,59 | 1 | 0 | 0 |
| P2 | 101 | 8,8 | 1 | 0 | 0 |
| P3 | 106 | 11,2 | 1 | 0 | 0 |
| P4 | 116 | 10,736 | 1 | 0 | 0 |
| P5 | 128 | 11 | 1 | 0 | 0 |
| P6 | 133 | 12 | 1 | 0 | 0 |
| P7 | 140 | 18,1 | 1 | 0 | 0 |
| P8 | 191 | 29,8 | 1 | 0 | 0 |
| P9 | 206 | 25 | 1 | 0 | 0 |

and no pseudo-alignments were returned.
Table 5.3 presents results compared with the other two algorithms. All algorithms were able to complete all alignments and compute the correct fitness value. More significant are results about computation time. Consider the graph in Figure 5.3, where the three different computation times are depicted: the Greedy approach is in blue, the classic approach (i.e. without using decomposition) is in green, and the Recomposition algorithm is in gray. It is evident that the classic approach is faster than the other two methods. Regarding the two decomposition-based approaches, the time of Greedy decomposition is better for only a few nets: *P2*, *P5*, *P7*, *P9*. However, we expected that the classical approach was the faster one; since the decomposition effort is not counter-balanced by an actual reduction of the complexity to compute alignments.

### 5.2.2 Noisy Scenario

In this section, we present Conformance Checking results for logs with noise. This means that there are discrepancies between the modeled and observed behavior such that the fitness value is less than 1. Similar to previous experiments, the goals are to compared computation times and fitness values with the two existing approaches.

For comparison purposes, the processes and models are the same as in the previous section, but noise is included in the logs during their generation, as discussed in the section on preliminary steps (Section 5.1). Each algorithm is tested with the introduction of noise of 6% and 14% for each process.

In each experiment, a 30-minutes time limit is set, and replays still running after the time limit are stopped and deemed infeasible. Both the Greedy Algorithm and Recomposing Replay produce a number of pseudo-alignments. During fitness analysis,

**Table 5.3:** Results about computation time and fitness between three algorithms in noiseless scenario.

| | Greedy approach | | Classic | | Recomposition | |
|---|---|---|---|---|---|---|
| **Name** | **Time(s)** | **Fitness** | **Time(s)** | **Fitness** | **Time(s)** | **Fitness** |
| P0 | 5,1 | 1 | 2 | 1 | 2,287 | 1 |
| P1 | 6,59 | 1 | 2 | 1 | 5,54 | 1 |
| P2 | 8,8 | 1 | 2 | 1 | 13,2 | 1 |
| P3 | 11,2 | 1 | 1 | 1 | 5,98 | 1 |
| P4 | 10,736 | 1 | 1 | 1 | 9,95 | 1 |
| P5 | 11 | 1 | 1 | 1 | 20,89 | 1 |
| P6 | 12 | 1 | 1 | 1 | 4,3 | 1 |
| P7 | 18,1 | 1 | 10 | 1 | 38,2 | 1 |
| P8 | 29,8 | 1 | 16 | 1 | 23,82 | 1 |
| P9 | 25 | 1 | 6 | 1 | 45,4 | 1 |

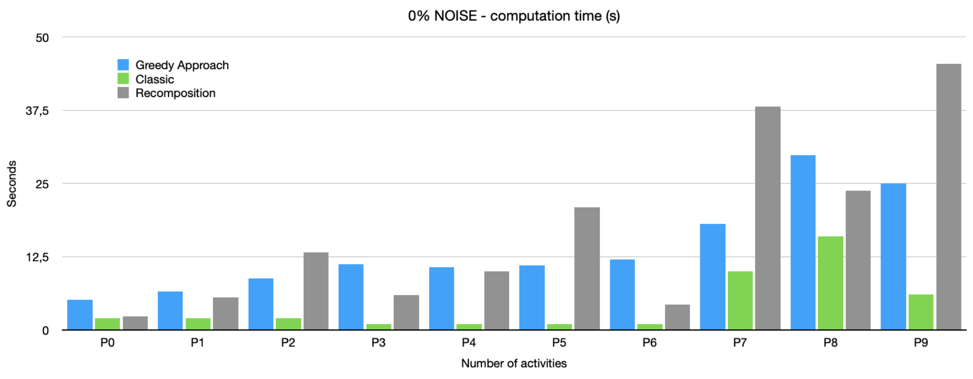


**Figure 5.3:** Graph representing computation time for each algorithm in noiseless scenario. The processes are ordered by number of activities. For each of them the time expressed in seconds are compared with three methods.

we remove those traces from event log, ensuring that every algorithm computes the fitness. The percentage of sub-alignments computed compared to the number of traces that present conflicts for the Greedy Approach is reported in Table 5.4. Additionally, the number of traces presenting conflicts and requiring the algorithm to start the search again to resolve disagreements is reported.

The classic approach computes exact results for any kind of traces and noise (expect for not reliable alignments, but they are not present in our experiments). The graph presented in Figure 5.4 shows the percentage of pseudo-alignments computed by the Greedy Approach and Recomposition Replay. Note that the percentage of traces whose

**Table 5.4:** Results about Greedy Approach for Decomposed Replay

| Greedy Approach | | | | | |
|---|---|---|---|---|---|
| **Name** | **Noise** | **Activities** | **Conflicts** | **Pseudo Alignments** | **Solved Alig.** |
| P0 | 6% | 43 | 832 | 19 | 47 |
| P0 | 14% | 43 | 976 | 64 | 141 |
| P1 | 6% | 67 | 811 | 23 | 90 |
| P1 | 14% | 67 | 858 | 55 | 154 |
| P2 | 6% | 101 | 924 | 125 | 104 |
| P2 | 14% | 101 | 993 | 373 | 82 |
| P3 | 6% | 106 | 850 | 185 | 118 |
| P3 | 14% | 106 | 1000 | 301 | 127 |
| P4 | 6% | 116 | 547 | 87 | 61 |
| P4 | 14% | 116 | 536 | 168 | 153 |
| P5 | 6% | 128 | 915 | 241 | 97 |
| P5 | 14% | 128 | 997 | 500 | 73 |
| P6 | 6% | 133 | 902 | 235 | 111 |
| P6 | 14% | 133 | 994 | 410 | 105 |
| P7 | 6% | 140 | 989 | 201 | 56 |
| P7 | 14% | 140 | 1000 | 717 | 73 |
| P8 | 6% | 191 | 997 | 17 | 128 |
| P8 | 14% | 191 | 1000 | 653 | 34 |
| P9 | 6% | 206 | 519 | 451 | 26 |
| P9 | 14% | 206 | 1000 | 800 | 27 |

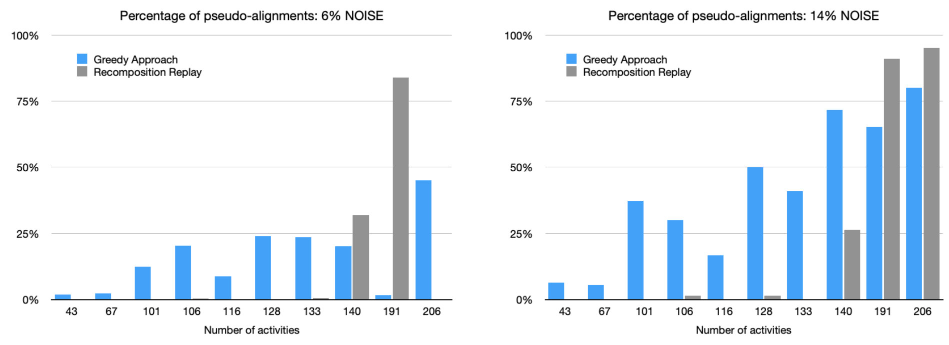


**Figure 5.4:** Graphs representing percentage of pseudo-alignments computed by Greedy Approach and Recomposition Replay. The first graph presents the traces with 6% of noise, the second one presents the traces with 14% of noise. With complex nets and more noise, Recomposition Replay results touch almost the 100% of pseudo-alignments.

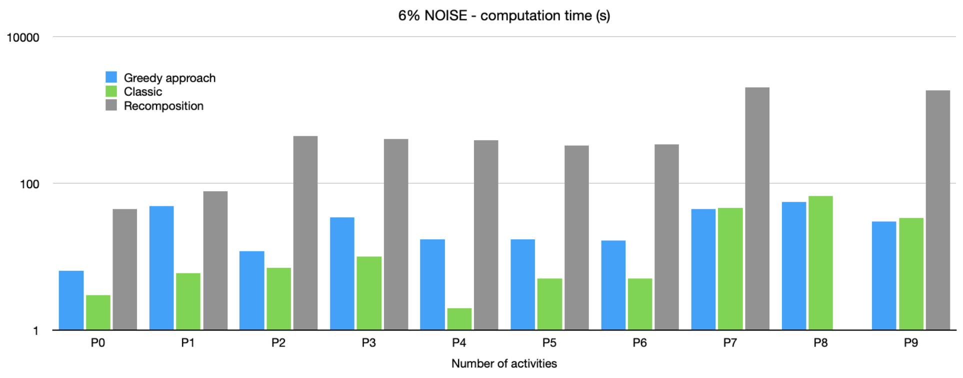


**Figure 5.5:** Graph representing computation time for each algorithm in a scenario with noise equal to 6%. The processes are ordered by number of activities. For each of them the time expressed in seconds are compared with three methods. Note that times are shown in a logarithmic scale.

reached the threshold (i.e. the fitness is not computed) increases with the growth of model and noise. Both of two algorithms are not able to compute alignments for all traces in event logs. In the next section we discuss these results and how to improve our algorithm.

### 6% noise

First, we report the computation times and fitness for the three algorithms using traces with a maximum noise of 6%. The results of our experiments are shown in Table 5.5. Considering computation time with the help of graph in 5.5, computation times are shown on a logarithmic scale. Classic approach is the faster algorithm, except for the last three models (more complex models) where our Greedy Approach takes slightly less time. There are clear performance gains from adopting Greedy Approach instead

**Table 5.5:** Results about computation time and fitness between three algorithms with noise=6%

| | Greedy approach | | Classic | | Recomposition | |
|---|---|---|---|---|---|---|
| **Name** | **Time(s)** | **Fitness** | **Time(s)** | **Fitness** | **Time(s)** | **Fitness** |
| P0 | 6,4 | 0,91 | 3 | 0,91 | 44,81 | 0,91 |
| P1 | 49 | 0,91 | 6 | 0,91 | 78,5 | 0,91 |
| P2 | 12 | 0,89 | 7 | 0,89 | 444,39 | 0,89 |
| P3 | 34,4 | 0,87 | 10 | 0,87 | 405 | 0,87 |
| P4 | 17,2 | 0,83 | 2 | 0,83 | 389 | 0,83 |
| P5 | 17,3 | 0,90 | 5 | 0,90 | 332 | 0,90 |
| P6 | 16,5 | 0,88 | 5 | 0,88 | 344,5 | 0,88 |
| P7 | 30 | 0,93 | 34 | 0,93 | 1875,2 | 0,93 |
| P8 | 44,7 | 0,92 | 46 | 0,93 | 2037 | 0,93 |
| P9 | 56,1 | 0,90 | 68 | 0,90 | ERROR | ERROR |

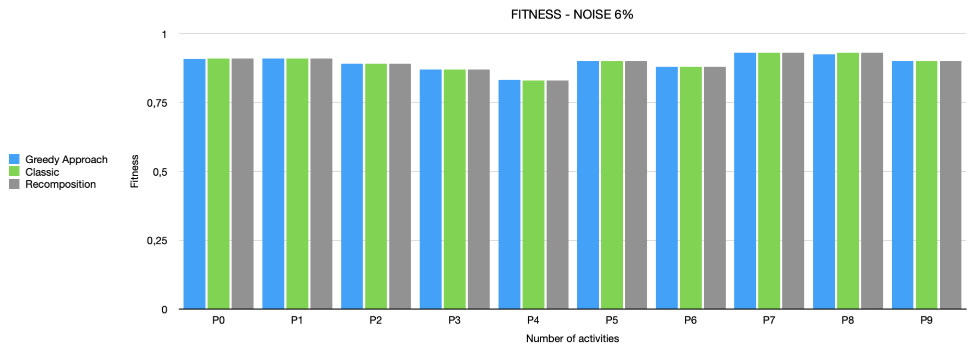


**Figure 5.6:** Graph representing fitness values for each algorithm in a scenario with noise equal to 6%. The processes are ordered by number of activities. The fitness are computed for decomposition approaches using just the traces which produce alignment (i.e. exact fitness can be calculated). Note that all three techniques return the same results for each model.

of Recomposition Replay.
Recomposition Replay was not able to compute alignments for Process *P9* and produced an unexpected error.

The graph in Figure 5.6 shows fitness values between the three algorithms. Recall that the metric is calculated only for traces for which results are alignments for both decomposition-based algorithms. For each model, the three techniques are tested for fitness with just this traces.

It is evident that all three algorithms produce the same result for each model.

**Table 5.6:** Results about computation time and fitness between three algorithms with noise=14%

| | Greedy approach | | Classic | | Recomposition | |
|---|---|---|---|---|---|---|
| **Name** | **Time(s)** | **Fitness** | **Time(s)** | **Fitness** | **Time(s)** | **Fitness** |
| P0 | 7,71 | 0,84 | 3 | 0,84 | 108,71 | 0,84 |
| P1 | 93,11 | 0,91 | 18 | 0,93 | 314,6 | 0,93 |
| P2 | 20,60 | 0,79 | 15 | 0,79 | 575,2 | 0,79 |
| P3 | 401 | 0,70 | 26 | 0,70 | 462,8 | 0,70 |
| P4 | 40,61 | 0,85 | 3 | 0,85 | 458,13 | 0,85 |
| P5 | 22,9 | 0,79 | 10 | 0,79 | 388,4 | 0,79 |
| P6 | 21,00 | 0,73 | 13 | 0,73 | 469,77 | 0,73 |
| P7 | 240 | 0,86 | 174 | 0,86 | 613,00 | 0,86 |
| P8 | 77,18 | 0,85 | 373 | 0,86 | 2166,40 | 0,86 |
| P9 | 367 | 0,85 | 226 | 0,85 | 1863,00 | 0,85 |

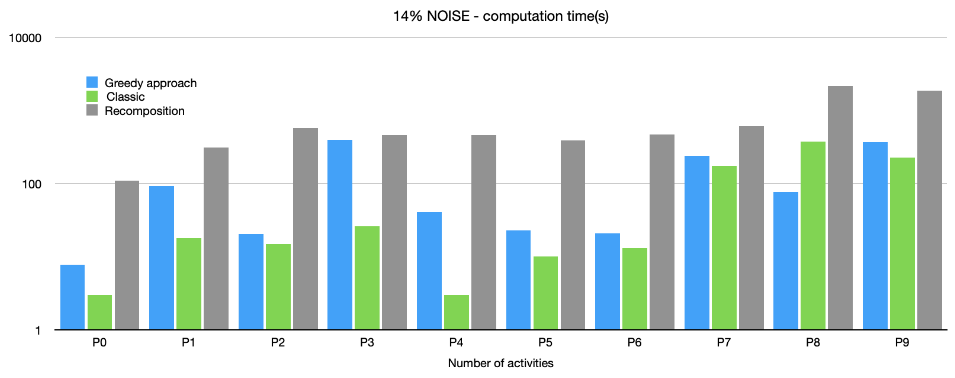


**Figure 5.7:** Graph representing computation time for each algorithm in a scenario with noise equal to 14%. The processes are ordered by number of activities. For each of them the time expressed in seconds are compared with three methods. Note that times are shown in a logarithmic scale.

As the classic approach returns only optimal alignments, using our dataset, both decomposition-based methods deliver optimal alignments too.

### 14% noise

At this point we increased the level of noise at 14%. The results of experiments are depicted in Table 5.6. In Figure 5.7 are shown the results about computation times between the three algorithms using a logarithmic scale. Even in this scenario, the classic approach is the faster algorithm except for *P8*, where the Greedy Approach is better in terms of performance. With the growth of noise, Recomposition Replay

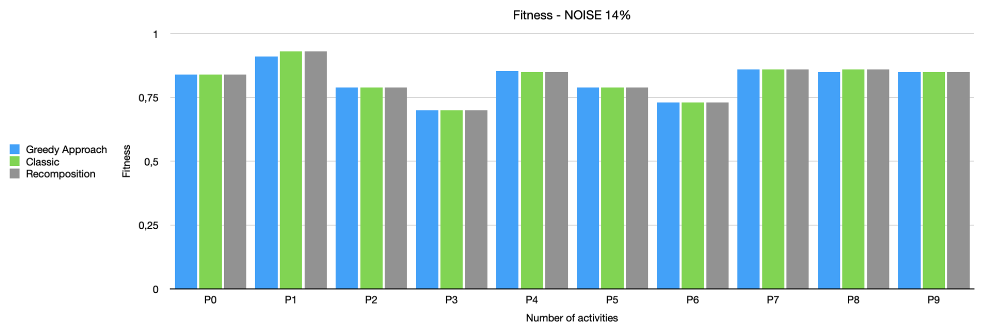


**Figure 5.8:** Graph representing fitness values for each algorithm in a scenario with noise equal to 14%. The processes are ordered by number of activities. The fitness are computed for decomposition approaches using just the traces which produce alignment (i.e. exact fitness can be calculated).

time increases as we expected (more conflicts mean the need for more recomposition iterations). Our Greedy method remains the best among decomposition approaches.

Regarding fitness, results are shown in Figure 5.8. Similarly to the previous scenario, the metric is computed only for those traces which produce alignments in all algorithms.

As in previous scenario, all methods deliver the same value, with the exception of models *P8* and *P1*. Here, Greedy approach produces a fitness smaller respect to the classic approach (which always generates the correct result). We were aware of this possibility as our algorithm does not guarantee optimal alignments if traces are not perfectly fitting.

## 5.3 Discussion and Future Works

This section discusses the results that were reported in the previous section. Considering the computation times of the three different solutions, the classic approach is the faster. However, in Recomposition Replay paper ([1]), the authors conclude that there are performance gains from adopting recomposition approach compared to the classic one. This is because the algorithm was not tested with the classic approach using the *A* iterative algorithm*, as it had not been introduced yet. The previous search algorithms were much slower, so Recomposition Replay was a valid and faster alternative. Another factor that negatively influenced Recomposition performance is the presence of noise. In [1] the noise introduced was not as high as in our tests. In fact, the resulting alignments in the nosy scenario in that paper have a fitness ranging from a maximum of 0,999 to a minimum of 0,947. Instead, the fitness computed in our experiments goes down to a minimum of 0,70. This means that as the noise increases, the number of disagreements also increases. Consequently, the number of recomposition iteration growths, and the Recomposition approach takes more time.

Recomposition Replay may be a valid alternative to the current classic technique in an environment of very large model (larger than the ones we used i.e. more than 200 activities) but with a not high quantity of noise. Note that in reality, processes could have even higher noise levels than the one we tested, so this scenario may not be so interesting.The Greedy Approach turns out to be faster than the Recomposition method but slower than classic approach. This means that the idea of maintaining the same decomposition of the model and trying to solve conflicts by just changing the subalignments is valid.However, the real issue with both decomposition-based techniques is the feasibility of computing exact fitness values; they can not compute all alignments, only pseudo-alignments, for many traces. As already mentioned, this might not be a problem if an estimation of the fitness is sufficient. Although, from our experiments, the classic technique is able to compute exact and optimal results within a shorter amount of time.

Regarding those traces for which Greedy approach provides the alignments, we have seen that this algorithm is valid. Hence, we believe that it may be a valid alternative of the Recomposition approach in the same domain described above: huge models and less noise.

Considering our experiments, we will study what may be the next steps in order to improve our algorithm.

### Improving search algorithm to produce replay

As described in Section 4.2, we use the same algorithm as the plug-in *"Replay a Log on Petri Net for All Optimal Alignments"*, but adapted and incorporates into our code. This algorithm provides the functionality of iteratively returning the next-best alignment and uses the standard *A** search algorithm to compute the replay.

A possible enhancement is to create an ad-hoc procedure that utilizes the faster *A* iterative* search algorithm to compute the replay and can provide all possible alignments, not just the optimal ones.

### Checking all possible combinations of subalignments

In the last column of Table 5.4 it is shown that by changing the starting subalignments of the search to find the right combination of subalignments, the algorithm is able to compute alignments for which the first search was not able to find.

Expanding this idea, a possible upgrade is to change how we search for the right combination by creating a searching tree where each node is a possible combination, and one or more subalignments are changed by computing the next-best. The root is the configuration of all optimal subalignments. In Figure 5.9 a naive representation of the searching tree is shown.
Each node may be expanded iteratively and can be chosen using the same heuristic (i.e. with a higher number of conflicts) or by inserting a random component. Using this representation, it is possible to revisit those combinations that have already been checked. If all combinations are tested, the right one will be selected eventually.

The implementation of this approach was not feasible within the timeframe of this project. The domain involves huge process models, consequently, they will have a vast number of possible combinations to test. This may impact memory usage, as all different combinations should be stored and linked to each other. We attempted to mimic this procedure in our improved version of the algorithm, where we tested more combinations by starting the search from different subalignments. However, the aim of

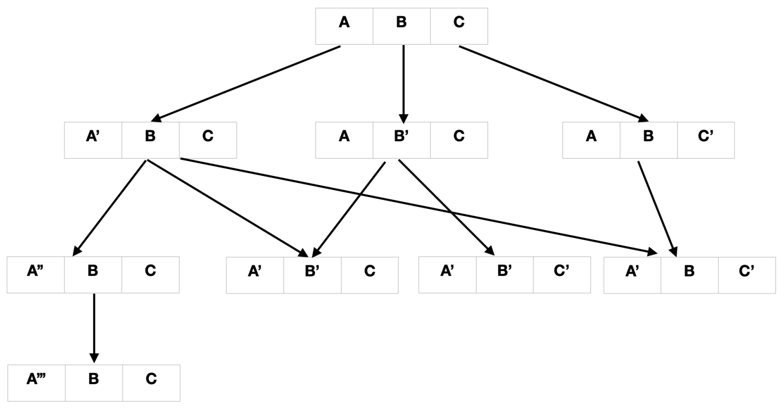


**Figure 5.9:** Naive representation of searching tree to check all possible subalignments combination. The subalignments are depicted using letters. In the root are present the combination of just optimal subalignments. Then subalignment A, B and C are selected sequentially to compute their next-best alignments. The search goes on by selecting one subalignments and computing its next-best.

this thesis was to test if it may be sufficient to use just next-best operation to solve conflict problems. Therefore, we limit the number of checked configurations to avoid exceeding both time and memory constraints.

# Chapter 6

# Conclusions

With the increasing of the amount of data, Process Mining researches are exploring new techniques to enhance existing methods. Conformance Checking analysis is one of the main fields effected by this growth. To improve the performance of existing algorithms, a new approach was proposed.
Decomposition-based approach has become a widely explored and well-documented branch of Process Mining research. In this thesis, we analyzed existing decomposition-based techniques such as *Pseudo-Alignments* and the more reliable *Recomposition Replay*. Before that, we formalized how to achieve a valid decomposition of a model and how this may be used in Conformance Checking domain.
*Pseudo-Alignments* algorithm produces just an estimation of the overall fitness, whereas *Recomposition Replay* guarantees exact fitness as subalignments computed are merged under the total border agreement condition. However, achieving this state may require many recomposition iterations, where the decomposition of the net changes, and replay is done again.

With our work, we aimed to find a trade-off between these two approaches. We wanted to resolve total border disagreements changing the subalignments instead of the decomposition of the net. In this way, we intended to save time, as the replay is done just one time, and subalignments are modified using the next-best subalignment operation.

This Conformance Checking approach was implemented as a plug-in in the ProM software. The last part of our work is dedicated to experiments in which we tested our algorithms compared to the classic approach and the recomposition method. The final results revealed both positive and negative aspects of the new algorithm. In our analysis, it emerged that both decomposition-based approaches became obsolete with the introduction of the new search algorithm *A* Iterative* to compute replay in classic plug-in. However, our future work could consist of implementing our decomposition method using this search algorithm instead of simple *A**.

The experiments show that in any case, this new technique can be a valid alternative to the Recomposition Replay as it significantly improves performance. This means that the idea of just modifying subalignments instead of changing the whole net decomposition is worthwhile of further exploration. One significant issue about our work is that it is not able to compute alignments for all traces in the log. This is because we limit the search of combinations of subalignments in order to save time and space in memory. One possible upgrade is to improve the search for the right combination of subalignments (i.e. one without conflicts) using the same idea of modifying subalignment by computing the next-best.

In conclusion, we proposed a valid alternative to the existing decomposition-based algorithm Recomposition Replay, which is also not able to compute the exact fitness for very complex problem and a considerable amount of noise except by employing a huge quantity of time. We are confident that with the suggested changes, this could also be a viable opponent for the younger classic algorithm.

# References

## Articles and Papers

[1] W. L. J. Leea, H. Verbeek, J. Munoz-Gamaa, et al. “Recomposing conformance: Closing the circle on decomposed alignment-based conformance checking in process mining.” In: *Information Science* (2018) (cit. on pp. 6, 23, 26, 27, 33, 37, 64).

[2] W. M. v. d. Aalst and C. Stahl. “Modeling Business Process: a Petri net-Oriented Approach”. In: (2011) (cit. on p. 7).

[3] W. M. v. d. Aalst. “Process Mining: data science in action”. In: (2016) (cit. on pp. 9–11, 14, 16, 17).

[4] B. F. van Dongen. “Efficiently Computing Alignments”. In: *Accepted for BPM* 11080 (2018) (cit. on pp. 20, 21).

[5] B. F. van Dongen. “Efficiently Computing Alignments: Algorithm and Data structures”. In: *Business Information Processing* 342 (2018) (cit. on p. 21).

[6] H. Völzer, J. Vanhatalo, and J. Koehler. “The refined process structure tree”. In: *Data and Knowledge Engineering* 68 (2009), pp. 793–818 (cit. on pp. 21, 28).

[7] W.M.P. v. d. Aalst. “Decomposing Petri nets for process mining: A generic approach.” In: *Distrib Parallel Databases* 31 (2013), pp. 471–507 (cit. on pp. 22, 24, 25, 27, 28).

[8] H.Völzer, A.Polyvyanyy, and J.Vanhatalo. “Simplified computation and generalization of the refined process structure tree”. In: *International Workshop on Web Services and Formal Methods* 6551 (2010), pp. 25–41 (cit. on p. 29).

[9] J. Carmona, J. Munoz-Gama, and W. M. v. d. Aalst. “Single-entry single-exit decomposed conformance checking”. In: *nformation Systems* 46 (2014), pp. 102–122 (cit. on pp. 29–31).

[10] W.M.P. v. d. Aalst and H.M.W. Verbeek. “Merging Alignments for Decomposed Replay”. In: *International Conference on Applications and Theory of Petri Nets and Concurrency* (2016) (cit. on pp. 31, 32, 37).

[11] W. M. v. d. Aalst, A. Adriansyah, A. K. A. d. Medeiros, et al. “Process Mining Manifesto”. In: *International conference on business process management* (2011), pp. 169–194.

[12] B.F. van Dongen, A. K. A Medeiros, H. M. W. Verbeek, et al. “The ProM Framework: A New Era in Process Mining Tool Support”. In: *Lecture Notes in Computer Science* 3536 (2005), pp. 444–454.

[13] H. M. W. Verbeek, H.M.W. Munoz Gama, and Aalst W.M.P. v. d. “Divide and conquer: a tool framework for supporting decomposed discovery in process mining”. In: *The Computer Journal* 60 (2017), pp. 1649–1674.

[14] W.M.P. v. d. Aalst. “Distributed Process Discovery and Conformance Checking”. In: *International Conference on Fundamental Approaches to Software Engineering* (2012).

[15] M. de Leoni, J. Munoz-Gama, J. Carmona, et al. “Decomposing Alignment-Based Conformance Checking of Data-Aware Process Models.” In: *Lecture Notes in Computer Science* 8841 (2014).

## Web Pages

[16] *Official ProM Software site*. URL: https://promtools.org.

[17] *Official PLG2 Software site*. URL: https://plg.processmining.it.

[18] *Official XES standard site*. URL: https://www.xes-standard.org.

[19] *Official Process Mining site*. URL: https://www.processmining.org.